**TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HCM**

**KHOA CÔNG NGHỆ THÔNG TIN**

****

**ĐỒ ÁN CUỐI KỲ  
MÔN HỌC: TRÍ TUỆ NHÂN TẠO**

**GIẢI VÀ MÔ PHỎNG TRÒ CHƠI SOKOBAN BẰNG CÁC THUẬT TOÁN TÌM KIẾM**

MÃ SỐ HP:241ARIN330585\_03CLC

GVHD: TS. Phan Thị Huyền Trang

SVTH: Nguyễn Hoài Bảo 22110108

Phan Phúc Hảo 22110134

Nguyễn Võ Cát Tường 22110263

**TP.HCM, ngày 02 Tháng 12 năm 2024**

**TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP.HCM**

**KHOA [CÔNG NGHỆ THÔNG TIN|CHẤT LƯỢNG CAO]**

**ĐỀ TÀI ĐỒ ÁN HỌC PHẦN**

TÊN HỌC PHẦN: **Trí tuệ nhân tạo**

MÃ SỐ HP: **241ARIN330585\_03CLC**

Tên đề tài: **GIẢI VÀ MÔ PHỎNG TRÒ CHƠI SOKOBAN BẰNG CÁC THUẬT TOÁN TÌM KIẾM**

GitHub Repository:[GitHub - MySokoban](https://github.com/ToiLaBao2004/MySokoban)

**NHẬN XÉT CỦA GIẢNG VIÊN**

**………………………………………………………..**

**………………………………………………………..**

**………………………………………………………..**

**………………………………………………………..**

**………………………………………………………..**

**………………………………………………………..**

**………………………………………………………..**

**………………………………………………………..**

**Ngày 02/12/2024**

**Giảng viên: Ký tên**

**TP.HCM, ngày 02 tháng 12 năm 2024**

# LỜI CẢM ƠN

Nhóm em xin gửi lời cảm ơn chân thành và sâu sắc đến giảng viên Phan Thị Huyền Trang, người đã không quản ngại thời gian và công sức để tận tâm giảng dạy, hướng dẫn và động viên chúng em trong suốt quá trình học tập và thực hiện đồ án môn Trí tuệ Nhân tạo.

Sự tận tình của cô không chỉ giúp chúng em có được nền tảng kiến thức lý thuyết vững chắc mà còn truyền cảm hứng để chúng em phát huy tinh thần sáng tạo và tư duy logic trong việc áp dụng các thuật toán trí tuệ nhân tạo vào thực tiễn. Trong suốt quá trình làm đồ án, cô không chỉ đóng vai trò là một người hướng dẫn mà còn là một người đồng hành, luôn sẵn sàng giải đáp mọi thắc mắc, hỗ trợ và khuyến khích chúng em vượt qua những khó khăn, thử thách.

Những lời khuyên quý báu của cô không chỉ giúp chúng em hoàn thiện đồ án một cách tốt nhất mà còn mang lại cho chúng em nhiều bài học ý nghĩa, cả trong học tập lẫn cuộc sống. Chính sự nhiệt huyết, chu đáo và sự quan tâm của cô đã tạo động lực mạnh mẽ để chúng em không ngừng cố gắng, hoàn thành nhiệm vụ với tinh thần trách nhiệm cao.

Ngoài ra, chúng em cũng xin gửi lời cảm ơn đến Nhà trường và Khoa Công nghệ Thông tin đã tạo điều kiện để chúng em có cơ hội học tập và thực hiện đồ án trong môi trường giáo dục chất lượng cao, được trang bị các nguồn tài nguyên học thuật phong phú và cơ sở vật chất hiện đại.

Cuối cùng, nhóm em cũng xin cảm ơn nhau – các thành viên trong nhóm, vì tinh thần đoàn kết, sự nỗ lực và phối hợp chặt chẽ để vượt qua những khó khăn trong quá trình thực hiện. Chúng em tin rằng những gì đã học được và trải nghiệm qua đồ án này sẽ là nền tảng vững chắc cho con đường học tập và sự nghiệp trong tương lai.

Một lần nữa, chúng em xin chân thành cảm ơn cô Phan Thị Huyền Trang vì những đóng góp to lớn trong hành trình học tập của chúng em.

**LỜI CAM KẾT**

Chúng em xin cam đoan rằng báo cáo này là kết quả của quá trình làm việc nghiêm túc, sáng tạo và độc lập của các thành viên trong nhóm. Toàn bộ nội dung trong báo cáo, bao gồm các phân tích, thiết kế, và triển khai, đều được thực hiện bởi chính các thành viên trong nhóm dưới sự hướng dẫn tận tình của giảng viên phụ trách.

Trong suốt quá trình nghiên cứu và thực hiện, chúng em đã tuân thủ đầy đủ các quy định về học thuật, đặc biệt là về đạo đức trong nghiên cứu. Chúng em không sao chép, trích dẫn tài liệu, mã nguồn hoặc ý tưởng của người khác mà không có sự ghi nguồn rõ ràng và phù hợp.

Nếu phát hiện bất kỳ hành vi nào liên quan đến sao chép hoặc vi phạm đạo văn trong quá trình thực hiện, chúng em xin hoàn toàn chịu trách nhiệm trước Nhà trường và giảng viên hướng dẫn.

Chúng em hy vọng rằng báo cáo này sẽ góp phần thể hiện năng lực học tập và tinh thần trách nhiệm của nhóm đối với môn học, đồng thời mang lại giá trị học thuật và thực tiễn hữu ích.

# DANH MỤC CÁC TỪ VIẾT TẮT

|  |  |
| --- | --- |
| **Từ viết tắt** | **Mô tả** |
| AI | Artificial Intelligence (Trí tuệ nhân tạo) |
| BFS | Breadth-First Search (Tìm kiếm theo chiều rộng) |
| DFS | Depth-First Search (Tìm kiếm theo chiều sâu) |
| A\* | A-Star Search (Tìm kiếm theo thuật toán A\*) |
| GUI | Graphical User Interface (Giao diện người dùng đồ họa) |
| CSV | Comma-Separated Values (Dữ liệu giá trị phân tách bằng dấu phẩy) |
| IDE | Integrated Development Environment (Môi trường phát triển tích hợp) |
| AI | Artificial Intelligence (Trí tuệ nhân tạo) |

**DANH MỤC CÁC THUẬT NGỮ**

|  |  |
| --- | --- |
| **Thuật ngữ** | **Định nghĩa** |
| Heuristic | Hàm đánh giá để dự đoán chi phí còn lại từ trạng thái hiện tại đến trạng thái đích. |
| State Space | |  | | --- | |  |  |  | | --- | | Không gian trạng thái, bao gồm tất cả các trạng thái có thể xảy ra trong bài toán tìm kiếm. | |
| Deadlock | Trạng thái bế tắc, nơi không thể đưa hộp đến vị trí đích. |

**MỤC LỤC**

[LỜI CẢM ƠN 3](#_Toc183962480)

[**LỜI CAM KẾT** 4](#_Toc183962481)

[DANH MỤC CÁC TỪ VIẾT TẮT 5](#_Toc183962482)

[**DANH MỤC CÁC THUẬT NGỮ** 5](#_Toc183962483)

[Chương 1: MỞ ĐẦU 9](#_Toc183962484)

[1.1. GIỚI THIỆU VỀ ĐỀ TÀI 9](#_Toc183962485)

[1.2. LÝ DO CHỌN ĐỀ TÀI 9](#_Toc183962486)

[1.3. PHÁT BIỂU BÀI TOÁN 10](#_Toc183962487)

[1.4. CÁC NỘI DUNG DỰ KIẾN THỰC HIỆN 11](#_Toc183962488)

[1.5. BỐ CỤC CỦA BÁO CÁO 11](#_Toc183962489)

[Chương 2: CƠ SỞ LÝ THUYẾT 13](#_Toc183962490)

[2.1. GIỚI THIỆU VỀ TRÍ TUỆ NHÂN TẠO 13](#_Toc183962491)

[2.2. TỔNG QUAN VỀ CÁC THUẬT TOÁN TÌM KIẾM 13](#_Toc183962492)

[2.2.1. Thuật toán tìm kiếm là gì? 13](#_Toc183962493)

[2.2.2. Phát biểu bài toán tìm kiếm 13](#_Toc183962494)

[2.2.3. Quy trình tìm kiếm trong không gian trạng thái 15](#_Toc183962495)

[2.2.4. Không gian trạng thái tường minh và không tường minh 15](#_Toc183962496)

[2.2.5. Thuật toán tìm kiếm tổng quát 16](#_Toc183962497)

[2.2.6. Các tiêu chuẩn đánh giá thuật toán tìm kiếm 17](#_Toc183962498)

[2.2.7. Thuật toán tìm kiếm mù (Uninformed Search) 18](#_Toc183962499)

[2.2.8. Thuật toán tìm kiếm có thông tin (Informed search) 20](#_Toc183962500)

[2.2.9. Thuật toán tìm kiếm cục bộ (Local search) 22](#_Toc183962501)

[2.2.10. Thuật toán tìm kiếm quay lui (Backtracking) 23](#_Toc183962502)

[2.3. GIỚI THIỆU CÁC THƯ VIỆN VÀ CÔNG CỤ THỰC HIỆN 24](#_Toc183962503)

[Chương 3: PHÂN TÍCH VÀ THIẾT KẾ GIẢI PHÁP 27](#_Toc183962504)

[3.1. MÔ TẢ BÀI TOÁN 27](#_Toc183962505)

[3.2. PHÂN TÍCH KHÔNG GIAN TRẠNG THÁI 27](#_Toc183962506)

[3.2.1. Không gian trạng thái 27](#_Toc183962507)

[3.2.2. Trạng thái bế tắc (Deadlock) 28](#_Toc183962508)

[3.3. THIẾT KẾ HỆ THỐNG 30](#_Toc183962509)

[3.3.1. Kiến trúc hệ thống 30](#_Toc183962510)

[3.3.2. Lưu đồ hệ thống 31](#_Toc183962511)

[3.3.3. Giao diện người dùng 32](#_Toc183962512)

[3.4. SƠ ĐỒ KHỐI / Ý TƯỞNG CÁC THUẬT TOÁN 34](#_Toc183962513)

[3.4.1. BFS 34](#_Toc183962514)

[3.4.2. DFS 35](#_Toc183962515)

[3.4.3. A\* 36](#_Toc183962516)

[3.4.4. Simulated Annealing 36](#_Toc183962517)

[3.4.5. Backtracking 37](#_Toc183962518)

[3.5. THIẾT KẾ CÁC HÀM DI CHUYỂN VÀ ĐẨY THÙNG 37](#_Toc183962519)

[3.5.1. Di chuyển 37](#_Toc183962520)

[3.5.2. Đẩy thùng 39](#_Toc183962521)

[3.5.3. Các bước di chuyển hợp lệ 42](#_Toc183962522)

[3.6. THIẾT KẾ HÀM TRÁNH DEADLOCK 43](#_Toc183962523)

[3.7. CÀI ĐẶT CHI TIẾT CÁC THUẬT TOÁN 45](#_Toc183962524)

[3.7.1. Breadth-First Search (BFS) 45](#_Toc183962525)

[3.7.2. Depth-First Search (DFS) 47](#_Toc183962526)

[3.7.3. A\* 49](#_Toc183962527)

[3.7.4. Backtracking 51](#_Toc183962528)

[3.7.5. Simulated Annealing 53](#_Toc183962529)

[Chương 4: THỰC NGHIỆM, ĐÁNH GIÁ, PHÂN TÍCH KẾT QUẢ 56](#_Toc183962530)

[4.1. MỤC TIÊU THỰC NGHIỆM 56](#_Toc183962531)

[4.2. MÔ PHỎNG QUÁ TRÌNH GIẢI SOKOBAN 56](#_Toc183962532)

[4.3. SO SÁNH CÁC THUẬT TOÁN 57](#_Toc183962533)

[Chương 5: KẾT LUẬN 65](#_Toc183962534)

[5.1. KẾT LUẬN 65](#_Toc183962535)

[5.2. HẠN CHẾ 65](#_Toc183962536)

[5.3. HƯỚNG PHÁT TRIỂN 66](#_Toc183962537)

[5.4. LỜI KẾT 66](#_Toc183962538)

[TÀI LIỆU THAM KHẢO 67](#_Toc183962539)

1. **MỞ ĐẦU**

## GIỚI THIỆU VỀ ĐỀ TÀI

Sokoban là một trò chơi giải đố kinh điển được phát triển vào những năm 1980 tại Nhật Bản. Từ "Sokoban" trong tiếng Nhật có nghĩa là “người quản lý kho hàng”, thể hiện đúng bản chất của trò chơi. Người chơi sẽ hóa thân thành một nhân vật có nhiệm vụ đẩy các hộp vào các vị trí được chỉ định trong một bản đồ giới hạn.

Điểm hấp dẫn của Sokoban nằm ở sự kết hợp giữa tính giải trí và tư duy logic. Các bản đồ trong trò chơi có độ khó tăng dần, yêu cầu người chơi phải lập kế hoạch cẩn thận để tránh rơi vào trạng thái bế tắc. Chính điều này đã biến Sokoban trở thành một bài toán lý thú trong lĩnh vực trí tuệ nhân tạo (AI), nơi các thuật toán tìm kiếm được áp dụng để giải quyết bài toán một cách hiệu quả.

Trong dự án này, nhóm nghiên cứu và triển khai các thuật toán trí tuệ nhân tạo như Breadth-First Search (BFS), Depth-First Search (DFS), A\* Search, Simulated Annealing, và Backtracking. Mục tiêu là không chỉ giải quyết bài toán mà còn đánh giá hiệu suất của từng thuật toán khi áp dụng trên các bản đồ với độ phức tạp khác nhau. Đồng thời, nhóm cũng xây dựng một hệ thống Sokoban hoàn chỉnh với giao diện trực quan, tích hợp khả năng hỗ trợ giải tự động bằng các thuật toán tìm kiếm.

## LÝ DO CHỌN ĐỀ TÀI

Sokoban không chỉ là một trò chơi giải trí phổ biến mà còn là một bài toán nghiên cứu kinh điển trong lĩnh vực trí tuệ nhân tạo. Trò chơi này mang lại cơ hội thực tiễn để áp dụng và kiểm chứng hiệu quả của các thuật toán tìm kiếm, giúp nhóm làm quen với việc triển khai các thuật toán trên các bài toán có tính ứng dụng cao. Đặc biệt, Sokoban có thể mô phỏng các bài toán tối ưu hóa trong thực tế, như quản lý kho hàng hoặc lập kế hoạch di chuyển.

Việc thực hiện đề tài Sokoban còn giúp nhóm phát triển kỹ năng tư duy logic và lập trình. Trò chơi đặt ra nhiều thách thức, bao gồm xử lý trạng thái bế tắc (deadlock), tối ưu hóa thuật toán để giảm thời gian thực hiện, và thiết kế giao diện đồ họa thân thiện với người dùng. Đây là cơ hội để nhóm rèn luyện các kỹ năng cần thiết cho việc phát triển các ứng dụng trí tuệ nhân tạo thực tiễn.

Ngoài ra, Sokoban mang tính linh hoạt cao, cho phép nhóm thử nghiệm nhiều ý tưởng sáng tạo. Nhóm có thể cải tiến các thuật toán để nâng cao hiệu suất hoặc thiết kế các bản đồ phức tạp hơn để kiểm tra khả năng giải quyết bài toán trong các tình huống đa dạng. Điều này không chỉ giúp nhóm hiểu sâu hơn về các thuật toán mà còn khuyến khích sự sáng tạo trong việc phát triển các giải pháp mới.

Cuối cùng, Sokoban là một đề tài phù hợp với trình độ của nhóm, vừa sức nhưng vẫn đủ thách thức để học hỏi và phát triển. Nó kết hợp hài hòa giữa yếu tố giải trí và học thuật, mang lại một môi trường nghiên cứu và thực nghiệm lý tưởng cho các thuật toán trí tuệ nhân tạo. Với tất cả những lý do trên, nhóm đánh giá rằng Sokoban là lựa chọn lý tưởng để thực hiện đồ án này.

## PHÁT BIỂU BÀI TOÁN

Trò chơi Sokoban có thể được biểu diễn như một bài toán tìm kiếm trong không gian trạng thái. Nhiệm vụ chính của bài toán là di chuyển các hộp (boxes) đến các vị trí đích được chỉ định trên bản đồ, tuân thủ các quy tắc cụ thể. Trạng thái ban đầu được xác định bởi vị trí của công nhân và các hộp trên bản đồ, trong khi trạng thái đích là khi tất cả các hộp được đặt đúng vào vị trí đích.

Trong quá trình thực hiện bài toán, công nhân chỉ có thể đẩy hộp, không được kéo và không được đẩy nhiều hơn một hộp cùng lúc. Các hành động di chuyển phải nằm trong phạm vi giới hạn của bản đồ, đảm bảo không tạo ra trạng thái bế tắc (deadlock). Mỗi bản đồ đại diện cho một thử thách khác nhau với độ phức tạp tăng dần, yêu cầu người chơi hoặc thuật toán phải tính toán đường đi hiệu quả nhất để đạt được mục tiêu.

Bài toán đặt ra yêu cầu tìm một chuỗi hành động (di chuyển và đẩy) để chuyển đổi từ trạng thái ban đầu sang trạng thái đích. Các hành động này cần được tối ưu hóa dựa trên các tiêu chí như số bước di chuyển, thời gian thực hiện, hoặc chi phí tổng thể. Trong dự án này, nhóm đã áp dụng các thuật toán trí tuệ nhân tạo như Breadth-First Search (BFS), Depth-First Search (DFS), A\* Search, Simulated Annealing, và Backtracking để giải quyết bài toán. Mỗi thuật toán được triển khai trên các bản đồ khác nhau nhằm đánh giá hiệu suất và khả năng tìm ra giải pháp tối ưu.

Với cách tiếp cận như trên, bài toán Sokoban không chỉ mang lại tính thách thức mà còn tạo cơ hội để nhóm thực nghiệm các thuật toán tìm kiếm trong trí tuệ nhân tạo, từ đó rút ra bài học về hiệu quả và ứng dụng của từng phương pháp.

## CÁC NỘI DUNG DỰ KIẾN THỰC HIỆN

* **Nghiên cứu lý thuyết**: Tìm hiểu các thuật toán tìm kiếm (BFS, DFS, A\*, Backtracking, Simulated Annealing) và các khái niệm liên quan như Deadlock và không gian trạng thái.
* **Phân tích bài toán**: Xác định luật chơi Sokoban, phân tích trạng thái và các yếu tố dẫn đến bế tắc.
* **Thiết kế và cài đặt**: Xây dựng hệ thống giải Sokoban, triển khai giao diện trực quan và các thuật toán.
* **Đánh giá hiệu suất**: So sánh các thuật toán dựa trên bước đi, số trạng thái duyệt và thời gian thực hiện.
* **Viết báo cáo**: Tổng hợp kết quả và đề xuất cải tiến.

## BỐ CỤC CỦA BÁO CÁO

Báo cáo được chia thành 5 chương, mỗi chương tập trung vào một nội dung chính như sau:

1. **Chương 1: Mở đầu**:

* Giới thiệu đề tài, lý do chọn đề tài, phát biểu bài toán, mục tiêu nghiên cứu, và phạm vi nghiên cứu.

1. **Chương 2: Cơ sở lý thuyết**:

* Trình bày các kiến thức nền tảng về trí tuệ nhân tạo, thuật toán tìm kiếm, không gian trạng thái, và các khái niệm liên quan.

1. **Chương 3: Phân tích và thiết kế giải pháp**:

* Phân tích bài toán Sokoban, thiết kế hệ thống và thuật toán, cùng với cách tiếp cận để giải quyết Deadlock.

1. **Chương 4: Kết quả thực nghiệm**:

* Trình bày dữ liệu thu thập, so sánh hiệu suất các thuật toán, và trực quan hóa kết quả.

1. **Chương 5: Kết luận và hướng phát triển**:

* Tổng kết các kết quả đạt được, bài học kinh nghiệm, và đề xuất cải tiến cho các nghiên cứu tiếp theo.

1. **CƠ SỞ LÝ THUYẾT**

## GIỚI THIỆU VỀ TRÍ TUỆ NHÂN TẠO

Trí tuệ nhân tạo (AI) là một lĩnh vực nghiên cứu thuộc khoa học máy tính, nhằm phát triển các hệ thống có khả năng thực hiện các nhiệm vụ phức tạp thường đòi hỏi trí tuệ của con người. Các nhiệm vụ này bao gồm: nhận dạng hình ảnh, xử lý ngôn ngữ tự nhiên, lập luận, và học tập dựa trên dữ liệu.

AI được phân loại thành:

* **AI yếu (Narrow AI)**: Hệ thống chuyên giải quyết một bài toán hoặc nhiệm vụ cụ thể, ví dụ: trợ lý ảo hoặc xe tự lái.
* **AI mạnh (General AI)**: Hệ thống có khả năng thực hiện mọi nhiệm vụ trí tuệ tương tự con người, tuy nhiên loại này hiện vẫn chỉ là lý thuyết.

Trong lịch sử phát triển, AI đã đi từ những ý tưởng ban đầu, như phép thử Turing, đến các ứng dụng thực tế, bao gồm hệ thống dự đoán, hỗ trợ y tế, và tối ưu hóa quy trình.

## TỔNG QUAN VỀ CÁC THUẬT TOÁN TÌM KIẾM

### Thuật toán tìm kiếm là gì?

Thuật toán tìm kiếm là các quy trình được thiết kế để khám phá và xác định con đường từ trạng thái ban đầu đến trạng thái mục tiêu trong không gian trạng thái. Nó được áp dụng rộng rãi trong các bài toán như giải đố, định tuyến đường đi, hoặc tối ưu hóa.

Các yếu tố chính của một bài toán tìm kiếm:

* **Trạng thái**: Tập hợp các điểm hoặc tình huống có thể xảy ra.
* **Hành động**: Các bước di chuyển giữa các trạng thái.
* **Mục tiêu**: Trạng thái cần đạt được.

### Phát biểu bài toán tìm kiếm

Bài toán tìm kiếm là một phương pháp giải quyết vấn đề bằng cách khám phá không gian trạng thái để tìm ra lời giải đáp ứng yêu cầu. Không gian trạng thái bao gồm tất cả các trạng thái liên quan đến bài toán và quá trình tìm kiếm sử dụng các phép chuyển đổi trạng thái để di chuyển từ trạng thái xuất phát đến trạng thái đích.

Bài toán tìm kiếm được mô tả thông qua năm thành phần cơ bản sau:

1. **Tập trạng thái (Q):** Đây là tập hợp toàn bộ các trạng thái có thể xảy ra trong bài toán, được gọi chung là không gian trạng thái. Mỗi trạng thái đại diện cho một tình huống cụ thể hoặc một điểm trong quá trình giải quyết vấn đề.
2. **Trạng thái xuất phát (S):** Là một hoặc nhiều trạng thái ban đầu từ đó thuật toán bắt đầu quá trình tìm kiếm. Các trạng thái xuất phát thường được xác định rõ ràng và nằm trong không gian trạng thái **Q**.
3. **Trạng thái đích (G):** Là một hoặc nhiều trạng thái mà bài toán yêu cầu đạt được. Trạng thái đích có thể được định nghĩa theo hai cách:
   1. **Tường minh:** Cụ thể chỉ ra trạng thái đích là gì (ví dụ: điểm đến cụ thể trên bản đồ).
   2. **Không tường minh:** Được mô tả qua các điều kiện mà trạng thái phải thỏa mãn (ví dụ: trạng thái chiến thắng trong trò chơi cờ vua là khi đối thủ bị chiếu hết).
4. **Các phép chuyển đổi (P):** Là tập hợp các hành động, toán tử hoặc phép biến đổi trạng thái. Mỗi phép chuyển đổi cho phép chuyển từ trạng thái hiện tại sang một hoặc nhiều trạng thái lân cận.
   1. Ký hiệu: **P(n)** là tập hợp các trạng thái có thể đạt được từ trạng thái **n**.
   2. Các trạng thái sinh ra từ một phép chuyển đổi được gọi là **trạng thái lân cận** hoặc **trạng thái hàng xóm**.
5. **Chi phí (c):** Là giá trị đại diện cho chi phí di chuyển giữa hai trạng thái, ký hiệu **c(x,y)**, trong đó **x** và **y** là hai trạng thái liền kề. Chi phí này có thể được xác định dựa trên yêu cầu của bài toán, chẳng hạn:
   1. **Quãng đường:** Chiều dài từ **x** đến **y**.
   2. **Thời gian:** Thời gian cần để di chuyển từ **x** đến **y**.
   3. **Năng lượng tiêu hao:** Chi phí cần để thực hiện hành động.

Tổng chi phí cho một chuỗi chuyển đổi từ trạng thái xuất phát đến trạng thái đích là tổng của tất cả các chi phí di chuyển giữa các trạng thái trung gian. Chi phí này được sử dụng để đánh giá chất lượng của lời giải, với lời giải có chi phí thấp nhất được coi là tối ưu.

### Quy trình tìm kiếm trong không gian trạng thái

1. **Khởi đầu:** Thuật toán bắt đầu từ trạng thái xuất phát **S**.
2. **Khám phá:** Các trạng thái hàng xóm được tạo ra bằng cách áp dụng các phép chuyển đổi **P** lên trạng thái hiện tại.
3. **Kiểm tra mục tiêu:** Sau mỗi lần chuyển đổi, thuật toán kiểm tra xem trạng thái mới có thỏa mãn điều kiện đích **G** hay không.
4. **Kết thúc:** Quá trình tìm kiếm dừng lại khi:
   1. Tìm thấy trạng thái đích (thỏa mãn điều kiện bài toán).
   2. Không còn trạng thái nào để khám phá (với bài toán không có giải).
5. **Trả lời:** Lời giải có thể là:
   1. **Chuỗi chuyển động:** Mô tả các hành động cần thực hiện để đạt đến trạng thái đích.
   2. **Trạng thái đích:** Trong một số bài toán (ví dụ: lập lịch), chỉ cần trả về trạng thái cuối cùng mà không cần chuỗi hành động.

### Không gian trạng thái tường minh và không tường minh

1. **Không gian trạng thái tường minh:** Các trạng thái được liệt kê đầy đủ và cụ thể.

Ví dụ: Trong bài toán tìm đường, mỗi địa điểm được biểu diễn dưới dạng một nút trong đồ thị.

1. **Không gian trạng thái không tường minh:** Không gian trạng thái được suy diễn từ trạng thái xuất phát và các phép chuyển đổi.

Trường hợp này, các trạng thái chỉ xuất hiện khi được tạo ra qua phép chuyển đổi và thường khó xác định toàn bộ không gian trạng thái trước khi tìm kiếm.

1. **Không gian trạng thái có thể là:**
   1. **Hữu hạn:** Chỉ bao gồm số lượng trạng thái giới hạn.
   2. **Vô hạn:** Bao gồm vô số trạng thái, như trong các bài toán suy diễn với số lượng phép chuyển đổi không giới hạn.

### Thuật toán tìm kiếm tổng quát

|  |
| --- |
| Graph\_Search(Q, S, G, P)  Input:  Q: Không gian trạng thái  S: Trạng thái xuất phát  G: Trạng thái đích  P: Tập các hành động  Output:  Chuỗi hành động hoặc trạng thái đích, hoặc thông báo không có giải pháp  1. Khởi tạo:  O ← {S} // Tập các nút biên  D ← ∅ // Tập các nút đã mở rộng (tập đóng)  2. While O ≠ ∅ do:  a. Lấy một nút n từ O  b. If n ∈ G then:  return đường đi tới n  c. Thêm n vào D  d. Sinh các trạng thái lân cận của n (P(n)) và thêm chúng vào O nếu chúng không thuộc D và O  3. Return "Không có giải pháp" |

Dưới đây là sơ đồ minh họa cây tìm kiếm:

* **Start**: Trạng thái xuất phát, tương ứng với gốc của cây.
* **Goal**: Trạng thái đích, là kết quả mà thuật toán tìm kiếm.
* **Các** **nhánh**: Đại diện cho hành động áp dụng trên trạng thái cha để tạo ra trạng thái con.

A diagram of a tree

Description automatically generated

Ưu điểm và hạn chế của thuật toán:

* Ưu điểm:
  + Tổng quát, áp dụng được cho nhiều bài toán khác nhau.
  + Dễ dàng tùy chỉnh thông qua cách mở rộng và kiểm tra trạng thái.
* Hạn chế:
  + Không quy định rõ thứ tự mở rộng nút, do đó hiệu quả phụ thuộc vào chiến lược cụ thể.
  + Có thể tiêu tốn nhiều tài nguyên nếu không gian trạng thái lớn.

### Các tiêu chuẩn đánh giá thuật toán tìm kiếm

Để đánh giá hiệu quả và khả năng áp dụng của các thuật toán tìm kiếm, bốn tiêu chuẩn chính thường được sử dụng như sau:

1. *Tính đầy đủ*: Thuật toán có đảm bảo tìm được lời giải nếu bài toán có lời giải không? Một thuật toán được gọi là đầy đủ nếu luôn tìm thấy lời giải trong trường hợp tồn tại, ngược lại, nó không đầy đủ.

2. *Tính tối ưu*: Thuật toán có thể tìm ra lời giải tốt nhất (với chi phí thấp nhất) khi bài toán có nhiều lời giải không? Tiêu chuẩn tối ưu thường dựa trên giá thành đường đi, thuật toán được coi là tối ưu nếu luôn tìm được lời giải có giá thành nhỏ nhất.

3. *Độ phức tạp tính toán*: Là khối lượng công việc mà thuật toán cần thực hiện để tìm ra lời giải. Điều này được đo lường thông qua số lượng trạng thái (nút) cần sinh ra và kiểm tra trước khi đạt được trạng thái đích.

4. *Yêu cầu bộ nhớ*: Là lượng bộ nhớ tối đa mà thuật toán cần để lưu trữ các trạng thái trong quá trình tìm kiếm. Điều này ảnh hưởng lớn đến khả năng áp dụng thuật toán trong các bài toán có không gian trạng thái lớn.

Các tiêu chuẩn đánh giá thuật toán tìm kiếm phụ thuộc vào độ khó (kích thước) của bài toán, đặc biệt là không gian trạng thái. Khi không gian trạng thái lớn, thuật toán cần nhiều thời gian và tài nguyên hơn để xử lý. Độ khó của bài toán có thể được xác định dựa trên ba tham số sau:

1.*Mức độ rẽ nhánh (b)*: Số lượng trạng thái con tối đa sinh ra từ một trạng thái cha.

2. *Độ sâu của lời giải (d)*: Số bước ngắn nhất từ trạng thái xuất phát đến trạng thái đích.

3. *Độ sâu tối đa của cây tìm kiếm (m)*: Chiều sâu lớn nhất mà cây tìm kiếm có thể đạt được.

### Thuật toán tìm kiếm mù (Uninformed Search)

Thuật toán tìm kiếm mù không sử dụng thông tin bên ngoài để quyết định lựa chọn tiếp theo trong quá trình tìm kiếm. Các thuật toán tìm kiếm mù sẽ mở rộng các trạng thái mà không có sự đánh giá ưu tiên nào, từ đó tìm kiếm tất cả các lựa chọn có thể cho đến khi tìm thấy giải pháp.

BFS là thuật toán tìm kiếm theo chiều rộng, nơi các trạng thái được mở rộng theo từng mức độ, tức là tất cả các trạng thái ở một mức độ hiện tại được xử lý trước khi chuyển sang mức độ tiếp theo. Thuật toán này đảm bảo tìm ra giải pháp tối ưu nếu không gian trạng thái không có trọng số. Tuy nhiên, BFS có nhược điểm là yêu cầu bộ nhớ lớn, đặc biệt khi không gian trạng thái rất lớn.

|  |
| --- |
| BFS(Q, S, G, P)  Input:  Q: Không gian trạng thái  S: Trạng thái xuất phát  G: Trạng thái đích  P: Tập các hành động  Output:  Chuỗi hành động hoặc trạng thái đích, hoặc thông báo "Không có lời giải"  1. Khởi tạo:  O ← {S} // Hàng đợi FIFO với trạng thái xuất phát  D ← ∅ // Tập các trạng thái đã được duyệt (tránh vòng lặp)  2. While O ≠ ∅ do:  a. Lấy nút đầu tiên n từ O và xóa n khỏi O  b. If n ∈ G then:  Return (đường đi tới n)  c. Add n vào D  d. For mỗi trạng thái con c thuộc P(n) do:  If c ∉ D và c ∉ O then:  Thêm c vào cuối O  3. Return "Không có lời giải" |

DFS là thuật toán tìm kiếm theo chiều sâu, trong đó một nhánh được mở rộng đến cùng trước khi quay lại thử các nhánh khác. Thuật toán này tiết kiệm bộ nhớ hơn so với BFS nhưng lại không đảm bảo tìm ra giải pháp tối ưu, đồng thời có thể rơi vào vòng lặp vô hạn nếu không có cơ chế dừng thích hợp.

|  |
| --- |
| DFS(Q, S, G, P)  Input:  Q: Không gian trạng thái  S: Trạng thái xuất phát  G: Tập trạng thái đích  P: Tập các hành động  Output:  Chuỗi hành động hoặc trạng thái đích, hoặc thông báo "Không có lời giải"  1. Khởi tạo:  O ← {S} // Ngăn xếp LIFO với trạng thái xuất phát  D ← ∅ // Tập các trạng thái đã được duyệt (tránh vòng lặp)  2. While O ≠ ∅ do:  a. Lấy nút đầu tiên n từ O và xóa n khỏi O  b. If n ∈ G then:  Return (đường đi tới n)  c. Add n vào D  d. For mỗi trạng thái con c thuộc P(n) do:  If c ∉ D và c ∉ O then:  Thêm c vào đầu O  3. Return "Không có lời giải" |

### Thuật toán tìm kiếm có thông tin (Informed search)

Thuật toán tìm kiếm có thông tin sử dụng hàm heuristic để đánh giá các trạng thái trong không gian tìm kiếm và ưu tiên mở rộng các trạng thái có khả năng dẫn đến mục tiêu tốt hơn. So với tìm kiếm mù, các thuật toán tìm kiếm có thông tin có thể giảm đáng kể thời gian tính toán và bộ nhớ bằng cách lựa chọn các bước đi tối ưu hơn.

A\* là thuật toán tìm kiếm phổ biến nhất trong tìm kiếm có thông tin. Thuật toán này kết hợp chi phí đi từ trạng thái hiện tại đến mục tiêu (hàm heuristic) với chi phí đã tiêu tốn (g), từ đó đưa ra một tổng chi phí (f = g + h) để lựa chọn bước tiếp theo. A\* đảm bảo tìm ra giải pháp tối ưu khi có một hàm heuristic phù hợp. Tuy nhiên, thuật toán này có thể tốn nhiều bộ nhớ và thời gian tính toán nếu không gian trạng thái rất lớn.

|  |
| --- |
| A\_Star(Q, S, G, P, h)  Input:  Q: Không gian trạng thái  S: Trạng thái xuất phát  G: Tập trạng thái đích  P: Tập các hành động  h: Hàm heuristic (ước lượng chi phí từ trạng thái hiện tại đến trạng thái đích)  Output:  Chuỗi hành động hoặc trạng thái đích, hoặc thông báo "Không có lời giải"  1. Khởi tạo:  O ← {S} // Tập mở  g(S) ← 0 // Chi phí từ trạng thái xuất phát đến S  f(S) ← g(S) + h(S) // Ước tính tổng chi phí  D ← ∅ // Tập đóng  2. While O ≠ ∅ do:  a. Lấy nút n từ O với giá trị f(n) nhỏ nhất và xóa n khỏi O  b. If n ∈ G then:  Return (đường đi tới n)  c. Thêm n vào D  d. For mỗi trạng thái con c thuộc P(n) do:  If c ∉ D và c ∉ O then:  g(c) ← g(n) + cost(n, c)  f(c) ← g(c) + h(c)  Thêm c vào O  Else If c ∈ O then:  If g(c) > g(n) + cost(n, c) then:  g(c) ← g(n) + cost(n, c)  f(c) ← g(c) + h(c)  End For  3. Return "Không có lời giải" |

### Thuật toán tìm kiếm cục bộ (Local search)

Thuật toán tìm kiếm cục bộ không tạo ra một cây tìm kiếm đầy đủ mà chỉ tìm kiếm trong không gian trạng thái từ một điểm xuất phát, tập trung vào việc tìm giải pháp gần nhất hoặc tối ưu nhất trong phạm vi hiện tại. Thuật toán này có thể rất hiệu quả trong các bài toán có không gian trạng thái lớn, nhưng đôi khi có thể bị mắc kẹt ở điểm tối ưu cục bộ và không tìm ra giải pháp tối ưu toàn cục.

Simulated Annealing là một thuật toán tìm kiếm cục bộ dựa trên nguyên lý mô phỏng quá trình làm lạnh của kim loại. Thuật toán này sẽ thử nghiệm các lựa chọn ngẫu nhiên trong không gian trạng thái và dần dần giảm mức độ thay đổi (giảm nhiệt độ) để tìm ra giải pháp tối ưu. Simulated Annealing rất hiệu quả trong việc tránh bị mắc kẹt ở các cực trị cục bộ, nhưng việc chọn hàm nhiệt độ phù hợp là rất quan trọng để đạt được hiệu quả tốt.

|  |
| --- |
| Simulated\_Annealing(Start, Goal, T, alpha, h)  Input:  Start: Trạng thái xuất phát  Goal: Tập trạng thái đích  T: Nhiệt độ ban đầu  alpha: Tỷ lệ giảm nhiệt độ (0 < alpha < 1)  h: Hàm heuristic đánh giá trạng thái  Output:  Trạng thái đích, hoặc thông báo "Không có lời giải"  1. Khởi tạo:  n ← Start  2. While T > 0 do:  a. If n ∈ Goal then:  Return (đường đi tới n)  b. Chọn ngẫu nhiên một trạng thái con c thuộc P(n)  c. ΔE ← h(n) - h(c)  d. If ΔE > 0 then:  n ← c // Chuyển đến trạng thái tốt hơn  Else If e^(ΔE / T) > random(0, 1) then:  n ← c // Chuyển đến trạng thái tệ hơn với xác suất giảm dần  e. Giảm nhiệt độ T ← T \* alpha  3. Return "Không có lời giải" |

### Thuật toán tìm kiếm quay lui (Backtracking)

Backtracking là một kỹ thuật giải quyết bài toán bằng cách thử nghiệm từng bước. Khi một lựa chọn không dẫn đến giải pháp hợp lệ, thuật toán sẽ quay lại bước trước đó và thử một lựa chọn khác. Kỹ thuật này thường được sử dụng trong các bài toán tổ hợp, như bài toán nữ hoàng cờ vua hoặc Sudoku, nơi có thể phải thử nhiều phương án khác nhau và quay lại khi gặp phải một tình huống không hợp lệ.

|  |
| --- |
| Backtracking(State, Goal, P)  Input:  State: Trạng thái hiện tại  Goal: Tập trạng thái đích  P: Tập các hành động  Output:  Chuỗi hành động hoặc trạng thái đích, hoặc thông báo "Không có lời giải"  1. If State ∈ Goal then:  Return (đường đi tới State)  2. For mỗi hành động a thuộc P(State) do:  Next ← Apply(State, a) // Sinh trạng thái mới  If Next hợp lệ then:  Result ← Backtracking(Next, Goal, P)  If Result ≠ "Không có lời giải" then:  Return Result  3. Return "Không có lời giải" |

## GIỚI THIỆU CÁC THƯ VIỆN VÀ CÔNG CỤ THỰC HIỆN

Dự án *GIẢI VÀ MÔ PHỎNG TRÒ CHƠI SOKOBAN BẰNG CÁC THUẬT TOÁN TÌM KIẾM* được phát triển bằng Python, sử dụng nhiều thư viện và công cụ hỗ trợ để xây dựng trò chơi và phân tích dữ liệu. Các thư viện và công cụ chính bao gồm:

**1. Pygame:**

* **Chức năng**: Pygame là một thư viện mạnh mẽ dành cho lập trình trò chơi 2D trong Python, hỗ trợ xử lý đồ họa, âm thanh, và tương tác sự kiện.
* **Vai trò trong dự án**: Pygame được sử dụng để xây dựng giao diện đồ họa của trò chơi Sokoban, bao gồm việc hiển thị các ô vuông, nhân vật, và các vật cản. Thư viện này cũng quản lý các sự kiện từ bàn phím, đảm bảo trò chơi phản hồi nhanh chóng với hành động của người chơi.

**2. Pandas:**

* **Chức năng**: Pandas cung cấp các công cụ để xử lý và phân tích dữ liệu có cấu trúc hoặc bán cấu trúc. Nó hỗ trợ đọc, ghi, và thao tác dữ liệu từ các nguồn như CSV.
* **Vai trò trong dự án**: Pandas được sử dụng để lưu trữ và phân tích kết quả của các thuật toán tìm kiếm được áp dụng trong trò chơi. Thông qua Pandas, nhóm có thể dễ dàng đánh giá hiệu suất của các thuật toán như BFS, DFS, A\*, SIMULATED ANNEALING và BACKTRACKING.

**3. Matplotlib:**

* **Chức năng:** Matplotlib là thư viện vẽ đồ thị phổ biến trong Python, cho phép trực quan hóa dữ liệu dưới dạng biểu đồ và hình ảnh.
* **Vai trò trong dự án:** Nhóm đã sử dụng Matplotlib để tạo các biểu đồ so sánh hiệu suất của các thuật toán tìm kiếm, giúp minh họa rõ ràng về số bước đi, thời gian thực hiện, và kết quả tối ưu.

**4. Jupyter Notebook:**

* **Chức năng:** Jupyter Notebook là một công cụ lập trình tương tác, hỗ trợ kết hợp mã Python, văn bản mô tả, và đồ thị trực quan trên cùng một tài liệu.
* **Vai trò trong dự án:** Jupyter Notebook được sử dụng để phân tích dữ liệu thu được từ các thuật toán tìm kiếm, trình bày kết quả và tạo báo cáo trực quan, giúp quá trình phân tích hiệu quả và dễ hiểu hơn.

**5. Tkinter:**

* **Chức năng:** Tkinter là thư viện GUI tiêu chuẩn của Python, giúp tạo ra các giao diện người dùng đơn giản.
* **Vai trò trong dự án:** Tkinter hỗ trợ trong việc xây dựng các cửa sổ giao diện bổ sung, chẳng hạn như bảng chọn mức độ hoặc bảng hiển thị kết quả, tăng trải nghiệm người dùng.

**6. PyCharm:**

* **Chức năng:** PyCharm là môi trường phát triển tích hợp (IDE) dành riêng cho Python, cung cấp các công cụ mạnh mẽ như gỡ lỗi, kiểm tra mã, và hỗ trợ lập trình.
* **Vai trò trong dự án:** PyCharm được sử dụng làm IDE chính để phát triển mã nguồn, quản lý dự án, và gỡ lỗi, giúp nhóm làm việc hiệu quả hơn trong suốt quá trình phát triển.

**7. Visual Studio Code:**

* **Chức năng:** Visual Studio Code là một IDE nhẹ, hỗ trợ nhiều ngôn ngữ lập trình, tích hợp Git, và cung cấp các tiện ích mở rộng phong phú.
* **Vai trò trong dự án:** Visual Studio Code được sử dụng song song với PyCharm, đặc biệt trong các tác vụ như quản lý mã nguồn trên GitHub và chỉnh sửa nhanh mã Python.

**8. GitHub:**

* **Chức năng**: Nền tảng quản lý mã nguồn trực tuyến hỗ trợ cộng tác và lưu trữ dự án.
* **Vai trò trong dự án**:
  + **Lưu trữ mã nguồn**: Toàn bộ mã nguồn và tài liệu của dự án được lưu trên GitHub, đảm bảo việc quản lý phiên bản và theo dõi thay đổi một cách hiệu quả.
  + **Chia sẻ và cộng tác**: Nhóm phát triển có thể làm việc cùng nhau và dễ dàng chia sẻ dự án với cộng đồng hoặc người dùng quan tâm.
  + **URL dự án**: Dự án được lưu tại [GitHub - MySokoban](https://github.com/ToiLaBao2004/MySokoban" \t "_new)

1. **PHÂN TÍCH VÀ THIẾT KẾ GIẢI PHÁP**

## 3.1. MÔ TẢ BÀI TOÁN

Sokoban là một trò chơi giải đố, nơi người chơi điều khiển một công nhân di chuyển trong một không gian giới hạn để đẩy các hộp vào đúng các vị trí đích được chỉ định. Trò chơi có vẻ đơn giản nhưng đòi hỏi tư duy chiến lược và lập kế hoạch cẩn thận để hoàn thành mục tiêu mà không rơi vào trạng thái bế tắc.

1. **Hành động**:

* Công nhân có thể di chuyển lên, xuống, trái, phải trong giới hạn bản đồ.
* Công nhân có thể đẩy một hộp nếu:
  + Vị trí phía sau hộp trống hoặc là vị trí đích.
  + Công nhân và hộp không bị cản bởi tường hoặc các hộp khác.

1. **Ràng buộc**:

* Không được đẩy hai hộp cùng lúc.
* Không được kéo hộp.
* Công nhân và hộp chỉ di chuyển trong không gian bản đồ.

1. **Mục tiêu**:

* Đưa tất cả các hộp vào đúng các vị trí đích với số bước đi hoặc chi phí tối ưu.

## 3.2. PHÂN TÍCH KHÔNG GIAN TRẠNG THÁI

### 3.2.1. Không gian trạng thái

Bài toán Sokoban có thể được mô tả như một bài toán tìm kiếm trong không gian trạng thái:

* **Trạng thái ban đầu:** Xác định vị trí của công nhân và các hộp trên bản đồ.
* **Trạng thái đích:** Khi tất cả các hộp nằm đúng các vị trí đích.
* **Phép chuyển đổi trạng thái:** Công nhân thực hiện các hành động di chuyển hoặc đẩy hộp.

Trạng thái được biểu diễn bằng một ma trận ký tự, trong đó mỗi ký tự đại diện cho một thành phần của bản đồ:

* ‘@’: Công nhân, nhân vật do người chơi điều khiển.
* ‘$’: Hộp, mục tiêu cần được di chuyển.
* ‘.’: Vị trí đích, nơi các hộp cần được đưa đến.
* ‘#’: Tường, cản trở di chuyển.
* ‘\*’: Hộp nằm trên vị trí đích.
* ‘ ’: Không gian trống, nơi công nhân có thể di chuyển.

Ví dụ level 1: Ví dụ level 10:

A screenshot of a computer

Description automatically generated A screenshot of a computer screen

Description automatically generated

**Chuyển đổi trạng thái**: Công nhân di chuyển hoặc đẩy hộp để thay đổi trạng thái:

* **Di chuyển**: Công nhân di chuyển đến ô trống.
* **Đẩy**: Công nhân đẩy một hộp nếu phía sau hộp là ô trống hoặc vị trí đích.

### 3.2.2. Trạng thái bế tắc (Deadlock)

Trạng thái bế tắc là trạng thái mà từ đó không thể đưa các hộp đến vị trí đích, bất kể chuỗi hành động nào được thực hiện. Trạng thái bế tắc thường xảy ra do các hộp bị kẹt ở những vị trí không thể đẩy hoặc không còn đường di chuyển hợp lệ.

Các loại trạng thái bế tắc phổ biến

1. Hộp bị kẹt trong góc:

* Hộp nằm ở các góc của bản đồ (góc trái, phải, trên, hoặc dưới) và không có cách nào để đẩy hộp ra khỏi góc.

Ví dụ:

A group of white hashtags on a black background

Description automatically generated

Trong ví dụ trên, hộp $ bị kẹt ở góc trên bên trái và không thể di chuyển.

1. Hộp bị chặn bởi các vật thể khác:

* Hộp bị bao quanh bởi các vật cản như tường (#) hoặc các hộp khác ($).

Ví dụ:

A group of white symbols on a black background

Description automatically generated

* Trong ví dụ trên, các hộp $ bị chặn và không thể di chuyển đến vị trí đích.

1. Vị trí đích không thể tiếp cận:

* Hộp không thể đến được vị trí đích do các hộp hoặc tường cản đường.

## 3.3. THIẾT KẾ HỆ THỐNG

### 3.3.1. Kiến trúc hệ thống

Hệ thống Sokoban được thiết kế với ba thành phần chính để đáp ứng cả nhu cầu giao diện người dùng và xử lý thuật toán:

1. Giao diện người dùng (UI):
   1. Được xây dựng bằng thư viện Pygame và Tkinter.
   2. Chức năng chính:
      * Hiển thị bản đồ trò chơi bao gồm các đối tượng: công nhân, hộp, vị trí đích, và tường.
      * Cho phép người chơi lựa chọn cấp độ trò chơi (Level Selection) và giao diện kết quả.
      * Hiển thị cảnh báo trạng thái bế tắc (Deadlock) hoặc thông báo chiến thắng khi hoàn thành.
2. Xử lý trò chơi (Game Logic):
   1. Xử lý các thao tác của người chơi như di chuyển công nhân, đẩy hộp, và kiểm tra trạng thái trò chơi.
   2. Tích hợp kiểm tra trạng thái bế tắc thông qua hàm *isDeadlock*.
   3. Xử lý logic xác định trạng thái hoàn thành trò chơi.
3. Thuật toán giải quyết (Solver):
   1. Triển khai các thuật toán tìm kiếm như BFS, DFS, A\*, Simulated Annealing và Backtracking.
   2. Đáp ứng yêu cầu tự động giải bài toán nếu người chơi yêu cầu.

### 3.3.2. Lưu đồ hệ thống

A diagram of a computer program

Description automatically generated

Lưu đồ hệ thống mô tả toàn bộ quy trình hoạt động của trò chơi, bao gồm các bước chính:

1. **Khởi tạo**:

* Người chơi lựa chọn cấp độ từ giao diện chính (Main UI).
* Bản đồ của cấp độ được tải và khởi tạo (Load Map).

1. **Vòng lặp trò chơi (Game Loop)**:

* Xử lý đầu vào của người chơi (Handle User Input), bao gồm di chuyển công nhân hoặc yêu cầu giải tự động.
* Kiểm tra trạng thái bế tắc (Check Deadlock):
  + Nếu phát hiện bế tắc, hiển thị cảnh báo (Display Deadlock Warning).
* Kiểm tra trạng thái hoàn thành (Check Game Completion):
  + Nếu hoàn thành, hiển thị thông báo chiến thắng (Display Win Message).

1. **Giải tự động**:

* Người chơi yêu cầu thuật toán giải tự động (Request Search Algorithm).
* Thuật toán Solver xử lý:
  + Nếu không tìm thấy lời giải, thông báo lỗi (No Solution).
  + Nếu tìm thấy lời giải, hiển thị đường đi tối ưu và thực thi (AI Player Path Execution).

### 3.3.3. Giao diện người dùng

**1. Giao diện Tkinter**

A screenshot of a video game

Description automatically generated

Khi khởi động chương trình thì giao diện chọn thuật toán sẽ hiện ra

A screenshot of a computer game

Description automatically generated

Có thể chọn các Level từ 1 đến 10 thông qua combobox

Bấm start để khởi chạy Pygame

**2. Giao diện Pygame**

Ví dụ Level 10 của game:

A screenshot of a video game

Description automatically generated

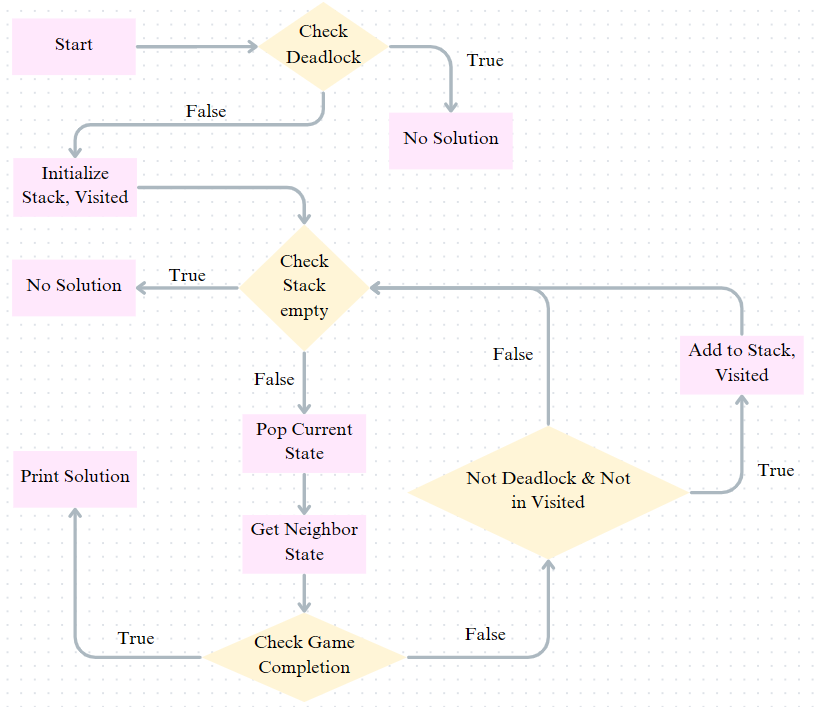
## 3.4. SƠ ĐỒ KHỐI / Ý TƯỞNG CÁC THUẬT TOÁN

### 3.4.1. BFS

A diagram of a solution

Description automatically generated

### 3.4.2. DFS



### 3.4.3. A\*

A diagram of a flowchart

Description automatically generated

### 3.4.4. Simulated Annealing

A diagram of a process

Description automatically generated

### 3.4.5. Backtracking

A diagram of a process

Description automatically generated

## 3.5. THIẾT KẾ CÁC HÀM DI CHUYỂN VÀ ĐẨY THÙNG

Lý do sử dụng tọa độ **(y, x)** thay vì **(x, y)**:

Trong lập trình với cấu trúc ma trận hoặc mảng hai chiều, quy ước phổ biến là:

* **Chỉ số hàng (y)** được sử dụng để truy cập dòng (row).
* **Chỉ số cột (x)** được sử dụng để truy cập cột (column).

Do đó, thứ tự **(y, x)** được ưu tiên vì nó phản ánh cách ma trận được truy cập trong Python. Ví dụ:

* Trong một ma trận **matrix**, phần tử tại dòng y và cột x được truy cập bằng **matrix[y][x]**.
* Nếu dùng **(x, y)** sẽ dẫn đến sự nhầm lẫn vì thứ tự này không khớp với cách lập trình ma trận thông thường.

### 3.5.1. Di chuyển

**1. Hàm lấy phần tử tại tọa độ (y, x):**

Hàm *getMatrixElement* được sử dụng để lấy giá trị của một phần tử tại tọa độ (y, x) trong ma trận. Đây là bước quan trọng để kiểm tra trạng thái hiện tại của vị trí trong bản đồ.

|  |
| --- |
| def getMatrixElement(self, y, x):  return self.matrix[y][x] |

Hàm *getElementNextStep* được sử dụng để lấy giá trị của một phần tử tại tọa độ (new\_y, new\_x) trong ma trận. Đây là bước quan trọng để kiểm tra trạng thái tiếp theo của công nhân sẽ di chuyển đến trong bản đồ.

|  |
| --- |
| def getElementNextStep(self, y, x):  new\_y, new\_x = self.workerPosition()[0] + y, self.workerPosition()[1] + x  return self.getMatrixElement(new\_y, new\_x) |

**2. Kiểm tra tính hợp lệ của bước di chuyển:**

Hàm *workerCanMove* kiểm tra xem công nhân có thể di chuyển đến ô (y, x) liền kề hay không. Công nhân chỉ được phép di chuyển đến các ô trống ‘ ’ hoặc vị trí đích ‘.’.

|  |
| --- |
| def workerCanMove(self, y, x):  return self.getElementNextStep(y, x) in [' ', '.'] |

Hàm này hoạt động như sau:

* Xác định tọa độ tiếp theo (new\_y, new\_x) dựa trên vị trí hiện tại của công nhân và hướng di chuyển (y, x).
* Sử dụng getMatrixElement để lấy giá trị phần tử tại (new\_y, new\_x).
* Kiểm tra nếu phần tử thuộc các trạng thái cho phép di chuyển là ô trống hay là vị trí đích.

**3. Hàm di chuyển:**

Hàm *move* xử lý việc di chuyển của công nhân trong bản đồ. Nó đảm bảo rằng công nhân chỉ được di chuyển đến các ô hợp lệ và cập nhật trạng thái bản đồ sau mỗi bước.

|  |
| --- |
| def move(self, y, x):  # Kiểm tra nếu công nhân có thể di chuyển  if self.workerCanMove(y, x):  worker\_position = self.workerPosition()  # Cập nhật vị trí mới cho công nhân  self.setMatrixElement(worker\_position[0] + y, worker\_position[1] + x, '@')  # Đặt lại vị trí cũ của công nhân  self.setMatrixElement(worker\_position[0], worker\_position[1], ' ')  # Phục hồi các vị trí đích bị ghi đè  for i, j in self.dockListPosition:  if self.getMatrixElement(i, j) not in ['\*', '@']:  self.setMatrixElement(i, j, '.') |

**Ví dụ hoạt động:**

Trước khi di chuyển: Công nhân di chuyển lên (1, 0):

**A group of white symbols on a black background

Description automatically generated A group of white symbols on a black background

Description automatically generated**

### 3.5.2. Đẩy thùng

**1. Kiểm tra tính hợp lệ của hành động đẩy:**

Hàm *workerCanPushBox* kiểm tra nếu công nhân có thể đẩy hộp theo hướng di chuyển (y, x):

* Sử dụng getElementNextStep để lấy giá trị tại ô kế tiếp (ô hộp) và ô tiếp theo nữa (ô phía sau hộp).
* Kiểm tra nếu:
  + Ô hộp nằm trong các trạng thái ‘\*’ (hộp trên đích) hoặc ‘$’ (hộp thông thường).
  + Ô phía sau hộp thuộc các trạng thái hợp lệ (‘ ’ hoặc ‘.’).

|  |
| --- |
| def workerCanPushBox(self, y, x):  return self.getElementNextStep(y, x) in ['\*', '$'] and self.getElementNextStep(y + y, x + x) in ['.', ' '] |

**Quy trình kiểm tra**:

1. Tìm tọa độ ô kế tiếp của công nhân (new\_y, new\_x).
2. Kiểm tra nếu ô kế tiếp chứa hộp ($) hoặc hộp trên đích (\*).
3. Kiểm tra nếu ô phía sau hộp là ô trống ( ) hoặc vị trí đích (.).

**2. Hàm đẩy thùng:**

Hàm *moveBox* chịu trách nhiệm cập nhật trạng thái của hộp và bản đồ khi công nhân thực hiện thao tác đẩy:

* Nhận các tham số:
  + y\_box, x\_box: Tọa độ hiện tại của hộp.
  + move\_y, move\_x: Hướng di chuyển.
* Cập nhật trạng thái của ô cũ và ô mới:
  + Nếu hộp di chuyển đến ô trống ( ), hộp được đặt là $.
  + Nếu hộp di chuyển đến vị trí đích (.), hộp được đặt là \*.

|  |
| --- |
| def moveBox(self, y\_box, x\_box, move\_y, move\_x):  # Lấy trạng thái hiện tại của hộp  box\_element = self.getMatrixElement(y\_box, x\_box)  # Lấy trạng thái của ô phía sau hộp  next\_box\_element = self.getMatrixElement(y\_box + move\_y, x\_box + move\_x)  # Xử lý di chuyển hộp  if box\_element == '$': # Hộp trên ô trống  if next\_box\_element == ' ':  self.setMatrixElement(y\_box, x\_box, ' ') # Làm trống ô cũ  self.setMatrixElement(y\_box + move\_y, x\_box + move\_x, '$') # Đặt hộp ở ô mới  elif next\_box\_element == '.':  self.setMatrixElement(y\_box, x\_box, ' ') # Làm trống ô cũ  self.setMatrixElement(y\_box + move\_y, x\_box + move\_x, '\*') # Đặt hộp trên đích  elif box\_element == '\*': # Hộp trên đích  if next\_box\_element == ' ':  self.setMatrixElement(y\_box, x\_box, '.') # Phục hồi ô đích  self.setMatrixElement(y\_box + move\_y, x\_box + move\_x, '$') # Đặt hộp ở ô mới  elif next\_box\_element == '.':  self.setMatrixElement(y\_box, x\_box, '.') # Phục hồi ô đích  self.setMatrixElement(y\_box + move\_y, x\_box + move\_x, '\*') # Đặt hộp trên đích mới |

**3. Kết hợp với hàm di chuyển:**

Hàm *move* tích hợp cả di chuyển và đẩy thùng:

* Kiểm tra nếu công nhân có thể đẩy hộp (workerCanPushBox).
* Nếu hợp lệ, gọi moveBox để cập nhật trạng thái hộp.

|  |
| --- |
| def move(self, y, x):  if self.workerCanMove(y, x):  # Di chuyển công nhân  worker\_position = self.workerPosition()  self.setMatrixElement(worker\_position[0] + y, worker\_position[1] + x, '@')  self.setMatrixElement(worker\_position[0], worker\_position[1], ' ')  elif self.workerCanPushBox(y, x):  # Đẩy hộp  worker\_position = self.workerPosition()  self.moveBox(worker\_position[0] + y, worker\_position[1] + x, y, x)  self.setMatrixElement(worker\_position[0] + y, worker\_position[1] + x, '@')  self.setMatrixElement(worker\_position[0], worker\_position[1], ' ') |

**Ví dụ hoạt động:**

Trước khi đẩy: Công nhân di chuyển qua phải (0, 1):

**A screenshot of a black background with white hashtags

Description automatically generated A group of white symbols on a black background

Description automatically generated**

### 3.5.3. Các bước di chuyển hợp lệ

Hàm *validMove* được sử dụng để xác định danh sách các bước di chuyển hợp lệ từ trạng thái hiện tại của công nhân trong trò chơi Sokoban. Nó kiểm tra các điều kiện di chuyển hoặc đẩy thùng và trả về danh sách các hướng di chuyển hợp lệ.

**Cách hoạt động**

Hàm kiểm tra bốn hướng di chuyển có thể xảy ra:

* **Lên** (U): Giảm tọa độ hàng (y - 1).
* **Xuống** (D): Tăng tọa độ hàng (y + 1).
* **Trái** (L): Giảm tọa độ cột (x - 1).
* **Phải** (R): Tăng tọa độ cột (x + 1).

**Đối với mỗi hướng di chuyển:**

1. **Kiểm tra tính hợp lệ của di chuyển**:
   * Sử dụng hàm *workerCanMove* để kiểm tra xem công nhân có thể di chuyển vào ô mục tiêu.
2. **Kiểm tra tính hợp lệ của đẩy thùng**:
   * Sử dụng hàm *workerCanPushBox* để kiểm tra xem công nhân có thể đẩy thùng vào ô phía sau thùng.

Nếu một trong hai điều kiện trên được thỏa mãn, hướng di chuyển sẽ được thêm vào danh sách kết quả.

|  |
| --- |
| def validMove(state):  moves = {  'U': (-1, 0),  'D': (1, 0),  'L': (0, -1),  'R': (0, 1)  }  valid\_moves = []  for step, (y, x) in moves.items():  if state.workerCanMove(y, x) or state.workerCanPushBox(y, x):  valid\_moves.append(step)  return valid\_moves |

## 3.6. THIẾT KẾ HÀM TRÁNH DEADLOCK

Trạng thái bế tắc (Deadlock) xảy ra khi các hộp trong trò chơi Sokoban bị đặt ở vị trí mà không thể di chuyển đến vị trí đích, bất kể chuỗi hành động nào được thực hiện. Việc phát hiện và tránh các trạng thái bế tắc giúp giảm không gian tìm kiếm và tăng hiệu quả của thuật toán giải.

|  |
| --- |
| def isDeadlock(state):  # Lấy danh sách vị trí các hộp  boxListPosition = state.boxPosition()  # Các điều kiện bế tắc  deadlock\_conditions = [  # Góc trên bên trái  lambda box\_y, box\_x: (  state.getMatrixElement(box\_y, box\_x - 1) in ['#', '$', '\*'] and  state.getMatrixElement(box\_y - 1, box\_x) in ['#', '$', '\*'] and  state.getMatrixElement(box\_y - 1, box\_x - 1) in ['#', '$', '\*']  ),  # Góc trên bên phải  lambda box\_y, box\_x: (  state.getMatrixElement(box\_y, box\_x + 1) in ['#', '$', '\*'] and  state.getMatrixElement(box\_y - 1, box\_x) in ['#', '$', '\*'] and  state.getMatrixElement(box\_y - 1, box\_x + 1) in ['#', '$', '\*']  ),  # Góc dưới bên trái  lambda box\_y, box\_x: (  state.getMatrixElement(box\_y, box\_x - 1) in ['#', '$', '\*'] and  state.getMatrixElement(box\_y + 1, box\_x) in ['#', '$', '\*'] and  state.getMatrixElement(box\_y + 1, box\_x - 1) in ['#', '$', '\*']  ),  # Góc dưới bên phải  lambda box\_y, box\_x: (  state.getMatrixElement(box\_y, box\_x + 1) in ['#', '$', '\*'] and  state.getMatrixElement(box\_y + 1, box\_x) in ['#', '$', '\*'] and  state.getMatrixElement(box\_y + 1, box\_x + 1) in ['#', '$', '\*']  ),  ]  # Kiểm tra từng hộp  for box in boxListPosition:  y, x = box  # Nếu bất kỳ điều kiện bế tắc nào thỏa mãn  if any(condition(y, x) for condition in deadlock\_conditions):  return True  # Nếu không phát hiện bế tắc  return False |

## 3.7. CÀI ĐẶT CHI TIẾT CÁC THUẬT TOÁN

### 3.7.1. Breadth-First Search (BFS)

**Nguyên lý:**

* BFS duyệt qua tất cả các trạng thái theo mức độ (chiều rộng) trước khi đi sâu vào trạng thái ở mức tiếp theo.
* BFS đảm bảo tìm ra lời giải ngắn nhất nếu tồn tại.

|  |
| --- |
| def bfs(game):  start = time.time()  node\_generated = 0  start\_state = copy.deepcopy(game)  node\_generated += 1  if isDeadlock(start\_state):  print("No Solution!")  return "NoSol"  queue = deque([start\_state])  visited = set()  visited.add(tuple(map(tuple, start\_state.getMatrix())))  print("Processing BFS......")  while queue:  currState = queue.popleft()  move = validMove(currState)  for step in move:  newState = copy.deepcopy(currState)  node\_generated += 1  if step == 'U':  newState.move(-1, 0)  elif step == 'D':  newState.move(1, 0)  elif step == 'L':  newState.move(0, -1)  elif step == 'R':  newState.move(0, 1)  newState.pathSolution += step  if newState.isComplete():  end = time.time()  print("Time to find solution:", round(end - start, 2), "seconds")  print("Number of visited nodes:", node\_generated)  print("Solution:", newState.pathSolution, "Number steps:", len(newState.pathSolution))  return newState.pathSolution  if (tuple(map(tuple, newState.getMatrix())) not in visited) and (not isDeadlock(newState)):  queue.append(newState)  visited.add(tuple(map(tuple, newState.getMatrix())))  print(node\_generated)  print("No Solution!")  return "NoSol" |

### 3.7.2. Depth-First Search (DFS)

Nguyên lý:

* DFS duyệt sâu vào từng nhánh trước khi quay lại các nhánh khác.
* DFS sử dụng ít bộ nhớ hơn BFS nhưng không đảm bảo tìm ra lời giải ngắn nhất.

|  |
| --- |
| def dfs(game):  start = time.time()  node\_generated = 0  start\_state = copy.deepcopy(game)  node\_generated += 1  if isDeadlock(start\_state):  print("No Solution!")  return "NoSol"  stack = [start\_state]  visited = set()  visited.add(tuple(map(tuple, start\_state.getMatrix())))  print("Processing DFS......")  while stack:  currState = stack.pop()  move = validMove(currState)  for step in move:  newState = copy.deepcopy(currState)  node\_generated += 1  if step == 'U':  newState.move(-1, 0)  elif step == 'D':  newState.move(1, 0)  elif step == 'L':  newState.move(0, -1)  elif step == 'R':  newState.move(0, 1)  newState.pathSolution += step  if newState.isComplete():  end = time.time()  print("Time to find solution:", round(end - start, 2), "seconds")  print("Number of visited nodes:", node\_generated)  print("Solution:", newState.pathSolution, "Number steps:", len(newState.pathSolution))  return newState.pathSolution  if (tuple(map(tuple, newState.getMatrix())) not in visited) and (not isDeadlock(newState)):  stack.append(newState)  visited.add(tuple(map(tuple, newState.getMatrix())))  print(node\_generated)  print("No Solution!")  return "NoSol" |

### 3.7.3. A\*

Nguyên lý:

* A\* kết hợp chi phí đã đi (g) và ước tính chi phí còn lại (h) để tìm trạng thái tốt nhất.
* Bằng cách tối ưu hàm f = g + h, A\* tìm lời giải hiệu quả và đảm bảo ngắn nhất nếu hàm heuristic phù hợp.

|  |
| --- |
| def astar(game):  start = time.time()  node\_generated = 0  start\_state = copy.deepcopy(game)  node\_generated += 1  start\_state.heuristic = worker\_toBox(start\_state) + box\_toDock(start\_state)  if isDeadlock(start\_state):  print("No Solution!")  return "NoSol"  open\_list = queue.PriorityQueue()  open\_list.put(start\_state)  close\_list = set()  print("Processing A\*......")  while not open\_list.empty():  cur\_state = open\_list.get()  move = validMove(cur\_state)  close\_list.add(tuple(map(tuple, cur\_state.getMatrix())))  for step in move:  new\_state = copy.deepcopy(cur\_state)  node\_generated += 1  if step == 'U':  new\_state.move(-1, 0)  elif step == 'D':  new\_state.move(1, 0)  elif step == 'L':  new\_state.move(0, -1)  elif step == 'R':  new\_state.move(0, 1)  new\_state.pathSolution += step  new\_state.heuristic = worker\_toBox(new\_state) + box\_toDock(new\_state)  if new\_state.isComplete():  end = time.time()  print("Time to find solution:", round(end - start, 2), "seconds")  print("Number of visited nodes:", node\_generated)  print("Solution:", new\_state.pathSolution, "Number steps:", len(new\_state.pathSolution))  return new\_state.pathSolution  if (tuple(map(tuple, new\_state.getMatrix())) not in close\_list) and not isDeadlock(new\_state):  open\_list.put(new\_state)  print(node\_generated)  print("No Solution!")  return "NoSol" |

### 3.7.4. Backtracking

**Nguyên lý**:

* Backtracking thử mọi khả năng di chuyển, quay lại các trạng thái trước đó nếu không tìm thấy giải pháp.

|  |
| --- |
| def backtracking(game):  start = time.time()  node\_generated = 0  start\_state = copy.deepcopy(game)  visited = set()  if isDeadlock(start\_state):  print("No Solution!")  return "NoSol"  print("Processing BACKTRACKING......")  def solve(state):  nonlocal node\_generated  # Nếu hoàn thành, trả về đường đi  if state.isComplete():  return state.pathSolution  # Lưu trạng thái vào visited  visited.add(tuple(map(tuple, state.getMatrix())))  # Thử từng bước đi khả dụng  for step in validMove(state):  newState = copy.deepcopy(state)  node\_generated += 1  # Thực hiện bước đi  if step == 'U':  newState.move(-1, 0)  elif step == 'D':  newState.move(1, 0)  elif step == 'L':  newState.move(0, -1)  elif step == 'R':  newState.move(0, 1)  newState.pathSolution += step  # Kiểm tra deadlock hoặc trạng thái đã ghé qua  if tuple(map(tuple, newState.getMatrix())) in visited or isDeadlock(newState):  continue  # Đệ quy tìm giải pháp từ trạng thái mới  solutionPath = solve(newState)  if solutionPath: # Nếu tìm được giải pháp, trả về  return solutionPath  # Không tìm thấy đường đi, quay lui  return None  # Bắt đầu giải bài toán  solution = solve(start\_state)  end = time.time()  if solution:  print("Time to find solution:", round(end - start, 2), "seconds")  print("Number of visited nodes:", node\_generated)  print("Solution:", solution, "Number steps:", len(solution))  return solution  else:  print("No Solution!")  return "NoSol" |

### 3.7.5. Simulated Annealing

**Nguyên lý**:

* Simulated Annealing chọn trạng thái ngẫu nhiên với xác suất giảm dần, cho phép chọn trạng thái kém hơn để tránh rơi vào cực trị cục bộ.

|  |
| --- |
| def simulated\_annealing(game, initial\_temperature=1000, cooling\_rate=0.95, max\_iterations=1000):  start = time.time()  node\_generated = 0  # Sao chép trạng thái ban đầu  current\_state = copy.deepcopy(game)  node\_generated += 1  # Đặt nhiệt độ ban đầu  temperature = initial\_temperature  # Đường đi tốt nhất và heuristic ban đầu  best\_path = ""  best\_heuristic = worker\_toBox(current\_state) + box\_toDock(current\_state)  print("Processing SIMULATED ANNEALING......")  for \_ in range(max\_iterations):  # Kiểm tra nếu đã hoàn thành  if current\_state.isComplete():  end = time.time()  print("Time to find solution:", round(end - start, 2), "seconds")  print("Number of visited nodes:", node\_generated)  print("Solution:", best\_path, "Number steps:", len(best\_path))  return best\_path  # Lấy tất cả các nước đi hợp lệ  moves = validMove(current\_state)  if not moves:  break  # Chọn ngẫu nhiên một bước đi  step = random.choice(moves)  next\_state = copy.deepcopy(current\_state)  node\_generated += 1  # Thực hiện bước đi  if step == 'U':  next\_state.move(-1, 0)  elif step == 'D':  next\_state.move(1, 0)  elif step == 'L':  next\_state.move(0, -1)  elif step == 'R':  next\_state.move(0, 1)  # Tính heuristic của trạng thái tiếp theo  next\_heuristic = worker\_toBox(next\_state) + box\_toDock(next\_state)  # Tính toán xác suất chọn trạng thái kém hơn  delta = next\_heuristic - best\_heuristic  if delta < 0 or random.uniform(0, 1) < math.exp(-delta / temperature):  current\_state = next\_state  best\_path += step  best\_heuristic = next\_heuristic  # Giảm nhiệt độ  temperature \*= cooling\_rate  print("No Solution Found!")  return "NoSol" |

1. **THỰC NGHIỆM, ĐÁNH GIÁ, PHÂN TÍCH KẾT QUẢ**

## 4.1. MỤC TIÊU THỰC NGHIỆM

Thực nghiệm được thực hiện nhằm đánh giá hiệu suất của các thuật toán tìm kiếm trong việc giải quyết trò chơi Sokoban. Các thuật toán được chọn để so sánh bao gồm: Breadth-First Search (BFS), Depth-First Search (DFS), A\* Search, Backtracking, và Simulated Annealing. Mục tiêu chính là kiểm tra khả năng của từng thuật toán trong việc tìm ra giải pháp tối ưu và hiệu quả khi áp dụng trên các cấp độ khác nhau của trò chơi.

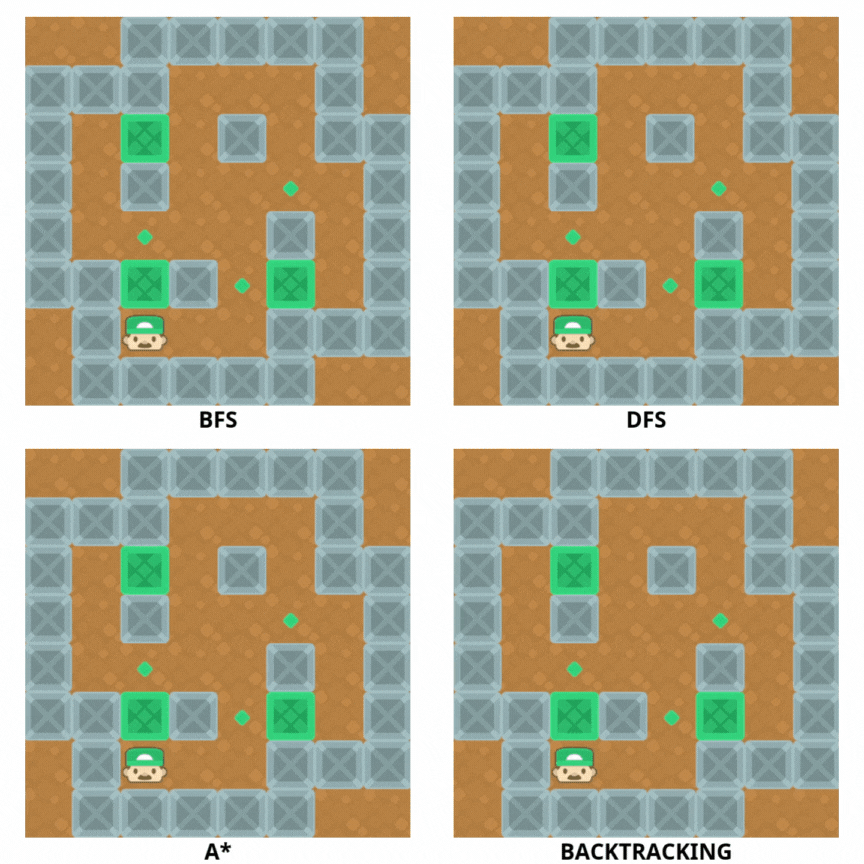
Các tiêu chí đánh giá hiệu suất được xác định như sau:

1. **Thời gian chạy (Time):** Đây là thời gian mà mỗi thuật toán cần để hoàn thành việc tìm kiếm giải pháp cho một cấp độ cụ thể. Thời gian chạy phản ánh khả năng tối ưu hóa tốc độ của thuật toán trong môi trường thực tế.
2. **Số bước đi (Steps):** Tiêu chí này đo lường số lượng hành động cần thực hiện để đạt đến trạng thái đích. Số bước đi ít hơn thường thể hiện một giải pháp tốt hơn, đặc biệt trong các bài toán yêu cầu tối ưu hóa lộ trình.
3. **Số trạng thái duyệt (Visited Nodes):** Tiêu chí này biểu thị số lượng trạng thái mà thuật toán phải duyệt qua trong quá trình tìm kiếm giải pháp. Số trạng thái duyệt càng thấp, thuật toán càng hiệu quả trong việc giới hạn không gian tìm kiếm, giúp tiết kiệm tài nguyên bộ nhớ và xử lý.

Các tiêu chí này không chỉ giúp so sánh các thuật toán về mặt lý thuyết mà còn mang lại góc nhìn thực tế về tính ứng dụng của chúng trong các bài toán tương tự, nơi tối ưu hóa thời gian và tài nguyên là những yêu cầu quan trọng.

## 4.2. MÔ PHỎNG QUÁ TRÌNH GIẢI SOKOBAN

Dưới đây sẽ là mô phỏng quá trình giải Sokoban của 4 thuật toán:



## 4.3. SO SÁNH CÁC THUẬT TOÁN

Dưới đây là dữ liệu thu thập được sau khi thực hiện mô phỏng 5 thuật toán với 10 level khác nhau của game sokoban từ dễ đến khó

A screen shot of a computer

Description automatically generated

**Bước 1: Đọc file dữ liệu**

A screenshot of a computer screen

Description automatically generated

Dữ liệu chứa **10 dòng** (tương ứng với 10 cấp độ) và **16 cột**, được mô tả chi tiết như sau:

* level: Cấp độ bài toán (1-10).
* bfs\_step, dfs\_step, astar\_step, backtracking\_step, simulatedannealing\_step: Số bước di chuyển cần thiết để giải bài toán cho từng thuật toán.
* bfs\_visited, dfs\_visited, astar\_visited, backtracking\_visited, simulatedannealing\_visited: Số trạng thái duyệt được trong quá trình tìm kiếm.
* bfs\_time, dfs\_time, astar\_time, backtracking\_time, simulatedannealing\_time: Thời gian chạy của từng thuật toán (đơn vị: giây).

**Bước 2: Kiểm tra các giá trị thiếu**

A screen shot of a computer program

Description automatically generated

Lưu ý rằng giá trị NaN ở Backtracking là do không có lời giải ở cấp độ 10.

Ghi nhận rằng tất cả giá trị NaN ở Simulated Annealing là do thuật toán không tìm được lời giải.

**Bước 3: Tiền xử lý dữ liệu & Trực quan hóa dữ liệu và nhận xét**

1. **Biểu đồ đường**

**A computer screen shot of a program code

Description automatically generated**

**A graph with lines and numbers

Description automatically generated with medium confidence**

Biểu đồ đường (Line Chart) so sánh số bước di chuyển (steps) giữa các thuật toán (BFS, DFS, A\*, Backtracking) qua các cấp độ (levels).

A graph with lines and numbers

Description automatically generated

Biểu đồ đường so sánh số trạng thái đã duyệt (visited states) giữa các thuật toán.

A graph with lines and numbers

Description automatically generated

Biểu đồ đường so sánh thời gian thực thi (execution time) giữa các thuật toán.

Các biểu đồ hiện tại chỉ so sánh các thuật toán trên từng cấp độ, nhưng không cung cấp một cái nhìn tổng quát, chẳng hạn như **hiệu suất trung bình** hoặc **tổng quan về các tiêu chí** (bước đi, trạng thái duyệt, thời gian) giữa các thuật toán.

1. **Biểu đồ cột**

Vì cần so sánh giá trị trung bình nên chúng ta phải drop level 10 là mức mà Backtracking không tìm thấy lời giải, vì nếu giữ lại chúng ta sẽ không có cái nhìn chính xác giữa các thuật toán.

A screen shot of a computer program

Description automatically generated

A graph of a bar chart

Description automatically generated with medium confidence

Biểu đồ cột so sánh số bước di chuyển trung bình từ trạng thái khởi tạo đến đích giữa các thuật toán.

Biểu đồ cho thấy BFS và A\* là hai thuật toán hiệu quả nhất về số bước trung bình, trong đó BFS thường tìm được giải pháp tối ưu. A\* cũng có kết quả tốt nhờ sử dụng hàm heuristic. DFS yêu cầu nhiều bước hơn do khám phá sâu trước, trong khi Backtracking có số bước trung bình cao nhất, phản ánh sự thiếu hiệu quả khi giải các bài toán phức tạp. Điều này cho thấy A\* và BFS phù hợp hơn cho các bài toán yêu cầu giải pháp nhanh và tối ưu.

A graph of different colored bars

Description automatically generated

Biểu đồ cột so sánh số trạng thái phải duyệt trung bình để tìm thấy trạng thái đích giữa các thuật toán.

Biểu đồ cho thấy BFS có số trạng thái trung bình được duyệt cao nhất trong các thuật toán, phản ánh cách tiếp cận toàn diện khi mở rộng toàn bộ trạng thái ở từng mức. DFS, A\*, và Backtracking có số trạng thái được duyệt thấp hơn đáng kể, trong đó A\* hiệu quả nhất nhờ sử dụng hàm heuristic để tối ưu hóa tìm kiếm. Điều này cho thấy A\* và Backtracking phù hợp hơn cho các bài toán lớn khi cần giảm số lượng trạng thái phải duyệt.

A graph of different colored squares

Description automatically generated

Biểu đồ cột so sánh thời gian thực hiện trung bình giữa các thuật toán.

Biểu đồ cho thấy BFS có thời gian thực thi trung bình cao nhất do duyệt qua toàn bộ trạng thái ở mỗi mức, dẫn đến tốn nhiều thời gian hơn. DFS, A\*, và Backtracking có thời gian thực thi thấp hơn, với DFS hoạt động nhanh nhờ mở rộng theo chiều sâu, trong khi A\* đạt hiệu quả nhờ sử dụng hàm heuristic. Backtracking có thời gian trung bình tương tự DFS và A\*, nhưng phụ thuộc nhiều vào cấu trúc của bài toán. Nhìn chung, A\* và DFS là các thuật toán cân bằng giữa hiệu quả và thời gian thực thi.

1. **KẾT LUẬN**

## 5.1. KẾT LUẬN

Dự án “Giải và Mô phỏng Trò chơi Sokoban bằng Các Thuật toán Tìm kiếm” đã hoàn thành các mục tiêu đề ra, bao gồm triển khai và đánh giá hiệu suất của các thuật toán tìm kiếm: BFS, DFS, A\*, Backtracking, và Simulated Annealing. Qua quá trình thực nghiệm, chúng tôi nhận thấy:

1. **Hiệu quả của các thuật toán**:

* **BFS**: Tìm kiếm giải pháp tối ưu nhất với số bước đi ít, nhưng đòi hỏi nhiều tài nguyên bộ nhớ và thời gian do duyệt qua toàn bộ trạng thái.
* **DFS**: Tiết kiệm bộ nhớ và thời gian nhưng không đảm bảo giải pháp tối ưu, dễ rơi vào trạng thái vòng lặp hoặc điểm không lối thoát.
* **A**\*: Tối ưu cả về thời gian và số trạng thái duyệt nhờ sử dụng hàm heuristic, phù hợp với các bài toán lớn và phức tạp.
* **Backtracking**: Kém hiệu quả ở các bài toán có độ phức tạp cao, nhưng phù hợp với bài toán nhỏ và có cấu trúc rõ ràng.
* **Simulated Annealing**: Không tìm được lời giải trong các cấp độ thử nghiệm, cho thấy hạn chế của thuật toán này khi áp dụng vào Sokoban.

1. **Trực quan hóa dữ liệu**:

* Biểu đồ so sánh đã minh họa rõ ràng sự khác biệt giữa các thuật toán về số bước đi, số trạng thái duyệt và thời gian thực hiện, giúp đánh giá hiệu quả từng phương pháp một cách toàn diện.

1. **Ứng dụng thực tế**:

* Các thuật toán tìm kiếm, đặc biệt là A\* và BFS, cho thấy tiềm năng ứng dụng trong các bài toán thực tế như tối ưu hóa lộ trình, quản lý kho hàng, và lập kế hoạch di chuyển.

## 5.2. HẠN CHẾ

1. Một số thuật toán (Simulated Annealing) không tìm được lời giải cho các cấp độ thử nghiệm, cho thấy cần điều chỉnh tham số hoặc cải thiện thuật toán.
2. Backtracking hoạt động không hiệu quả trên các bài toán phức tạp, dễ dẫn đến chi phí tính toán cao.
3. Mô phỏng chỉ tập trung vào 10 cấp độ, chưa thể bao quát hết các cấu trúc bản đồ phức tạp hơn của Sokoban.

## 5.3. HƯỚNG PHÁT TRIỂN

1. Một số thuật toán (Simulated Annealing) không tìm được lời giải cho các cấp độ thử nghiệm, cho thấy cần điều chỉnh tham số hoặc cải thiện thuật toán.
2. Backtracking hoạt động không hiệu quả trên các bài toán phức tạp, dễ dẫn đến chi phí tính toán cao.
3. Mô phỏng chỉ tập trung vào 10 cấp độ, chưa thể bao quát hết các cấu trúc bản đồ phức tạp hơn của Sokoban.

## 5.4. LỜI KẾT

Dự án không chỉ là một thử nghiệm về mặt kỹ thuật mà còn giúp nhóm hiểu sâu hơn về tư duy thuật toán và khả năng giải quyết vấn đề thực tiễn. Việc so sánh các thuật toán như BFS, DFS, A\*, Backtracking, và Simulated Annealing không chỉ giúp nhóm nắm bắt được ưu nhược điểm của từng phương pháp mà còn rút ra được bài học quý giá về cách áp dụng chúng vào từng bài toán cụ thể.

Chúng tôi tin rằng, với những kết quả đã đạt được, dự án có thể trở thành tiền đề cho những nghiên cứu sâu hơn trong các lĩnh vực liên quan như tự động hóa, robot điều hướng, và tối ưu hóa lộ trình. Đồng thời, đây cũng là minh chứng cho sự kết hợp giữa lý thuyết và thực tiễn trong việc giải quyết các bài toán phức tạp.

Chúng tôi hy vọng, với những cải tiến và phát triển trong tương lai, dự án sẽ không chỉ dừng lại ở bài toán Sokoban mà còn mở rộng phạm vi ứng dụng trong các bài toán thực tiễn khác, đóng góp vào sự phát triển của lĩnh vực Trí tuệ Nhân tạo và Tin học.

Trân trọng cảm ơn sự hỗ trợ từ giảng viên và đồng đội đã giúp chúng tôi hoàn thành dự án này!

# TÀI LIỆU THAM KHẢO

[1]. Phạm Vĩnh Lộc. (2021, October 17). *Demo Sokoban with BFS and A* [Video]. YouTube. <https://www.youtube.com/watch?v=bKK74HN4T9c&t=178s>

[2]. Từ Minh Phương. (2014). *Nhập môn trí tuệ nhân tạo*. Chương 1 và Chương 2. Học viện Công nghệ Bưu chính Viễn thông.

[3]. Kenney. (n.d.). *Sokoban Pack* [Digital image]. OpenGameArt. Licensed under CC-BY 3.0. <https://opengameart.org/content/sokoban-pack>