**A white rectangular frame with blue border

Description automatically generated**

**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

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

**KHOA ĐÀO TẠO QUỐC TẾ**

|  |  |
| --- | --- |
| ***Nhóm sinh viên thực hiện:*** | |
| Nguyễn Ngọc Mạnh | 21110781 |
| Võ Hoàng Tùng | 21110811 |
| Đặng Văn Thông | 19110056 |
| Nguyễn Ngọc Minh | 21110784 |
|  |  |

**A logo with hands holding a book and a flame

Description automatically generated**

**LỜI MỞ ĐẦU**

**TP. Hồ Chí Minh, tháng 11 năm 2023**

**Giảng viên: Trần Nhật Quang**

**Báo cáo cuối kỳ:**

**Artificial Intelligence**

Trong thời đại 4.0 hiện nay, trí tuệ nhân tạo (Artifical Intelligence) đóng vai trò quyết định sự thành công của các ngành và quốc gia. Sự phát triển mạnh mẽ của AI đã thay đổi cuộc sống của chúng ta, ảnh hưởng đến nhiều lĩnh vực như quân đội, y tế, giáo dục, kinh tế, tài chính, và nhiều ngành khác. Việc áp dụng AI để giải quyết các vấn đề xã hội và thúc đẩy phát triển kinh tế được khuyến khích và đầu tư lớn từ nhà nước.

Công nghệ thông tin, đặc biệt là trí tuệ nhân tạo, đang có ảnh hưởng mạnh mẽ đến sự phát triển của nhiều quốc gia, bao gồm cả Việt Nam. Đây trở thành yếu tố quyết định đến thành công hay thất bại của nhiều ngành. Sự phát triển của công nghệ thông tin, đặc biệt là AI, đang là xu hướng nổi bật và được ưu chuộng.

Chúng em xin cảm ơn thầy Trần Nhật Quang. Thầy đã chia sẻ nguồn kiến thức và cung cấp tài liệu hữu ích giúp chúng em nắm bắt kiến thức và tiến triển trong quá trình học tập và tìm hiểu. Chúng em trân trọng sự hỗ trợ đó và xin chân thành cảm ơn!

**ĐIỂM SỐ**

|  |  |  |  |
| --- | --- | --- | --- |
| **TIÊU CHÍ** | **NỘI DUNG** | **TRÌNH BÀY** | **TỔNG** |
| **ĐIỂM** |  |  |  |

**NHẬN XÉT**

*Ký tên*

**Trần Nhật Quang**

**MỤC LỤC**

[1.ĐẠO VĂN 1](#_Toc152234289)

[1.1 Đạo văn là gì ? 1](#_Toc152234290)

[1.2 Những điều nên/không nên làm khi làm bài báo cáo 1](#_Toc152234291)

[1.3 Cam kết 2](#_Toc152234292)

[2. TÌM HIỂU VỀ TRÍ TUỆ NHÂN TẠO 3](#_Toc152234293)

[2.1 Trí tuệ nhân tạo(Artificial Intelligence) là gì 3](#_Toc152234294)

[2.2 Lích sử hình thành của AI 3](#_Toc152234295)

[2.3 Đạo đức trí tuệ nhân tạo 4](#_Toc152234296)

[2.4 Cách tiếp cận của AI 6](#_Toc152234297)

[3. CÁC THUẬT TOÁN TÌM KIẾM CƠ BẢN 6](#_Toc152234298)

[3.1 Thuật toán tìm kiếm là gì ? 6](#_Toc152234299)

[3.2 Các bước cơ bản của một thuật toán tìm kiếm 7](#_Toc152234300)

[3.3 Tiêu chí đánh giá thuật toán 7](#_Toc152234301)

[3.4 Các thành phần của một vấn đề 7](#_Toc152234302)

[3.5 Conceptual algorithm (Thuật toán mẫu ) 8](#_Toc152234303)

[3.5.1 Conceptual algorithm #1 – TREE SEARCH 8](#_Toc152234304)

[3.5.2 Conceptual algorithm #2 GRAPH SEARCH 9](#_Toc152234305)

[3.6 Phân loại thuật toán tìm kiếm 10](#_Toc152234306)

[3.7 Cấu trúc dữ liệu 10](#_Toc152234307)

[3.7.1 Thành phần của một node 10](#_Toc152234308)

[3.7.2 Queue (hàng đợi) 10](#_Toc152234309)

[3.8 Uninformed search algorithm 11](#_Toc152234310)

[3.8.1 Breadth first search (BFS) 11](#_Toc152234311)

[3.8.2 Uniform cost search 12](#_Toc152234312)

[3.8.3 Depth first search(DFS) 13](#_Toc152234313)

[3.8.4 Depth limited search (DLS) 14](#_Toc152234314)

[3.8.5 Iterative deepening search 16](#_Toc152234315)

[3.9 Informed search algorithm 16](#_Toc152234316)

[3.9.1 Các hàm ước lượng quãng đường 16](#_Toc152234317)

[3.9.2 Ý tưởng chung của informed search 17](#_Toc152234318)

[3.9.3 Best first search 18](#_Toc152234319)

[3.9.4 A\* search 19](#_Toc152234320)

[3.9.5 Cách tạo ra heuristic funcion 20](#_Toc152234321)

[3.9.6 So sánh hàm heuristic 21](#_Toc152234322)

[4. LOCAL SEARCH – SEARCHING FOR GOAL STATE 21](#_Toc152234323)

[4.1 State space lanscape 21](#_Toc152234324)

[4.2 Local search 23](#_Toc152234325)

[4.3 Hill Climbing 23](#_Toc152234326)

[4.4 Các vấn đề của thuật toán Hill Climbing 24](#_Toc152234327)

[4.4.1 Quá nhiều successors 24](#_Toc152234328)

[4.4.2 Local optimum 25](#_Toc152234329)

[4.5 Thuật toán Simulated Annealing 26](#_Toc152234330)

[4.6 Searching in none-deterministic environments 27](#_Toc152234331)

[4.6.1 Tìm kiếm trong môi trường không xác định 27](#_Toc152234332)

[4.6.2 Thuật toán And-Or Graph Search 28](#_Toc152234333)

[4.7 Searching in partially observable environments 30](#_Toc152234334)

[4.7.1 No observation 30](#_Toc152234335)

[4.7.2 Partially observable 31](#_Toc152234336)

[4.8 Online search 32](#_Toc152234337)

[4.8.1 So sánh offline search và online search 32](#_Toc152234338)

[4.8.2 Competitive ratio 33](#_Toc152234339)

[4.8.3 Online depth first search agent 34](#_Toc152234340)

[4.8.4 Online A\* search 35](#_Toc152234341)

[4.8.5 Learning real- Time A\* agent 36](#_Toc152234342)

[4.9 Constraint Satisfactoin Problems 37](#_Toc152234343)

[4.9.1 Constraint satisfaction problems (CSP) 37](#_Toc152234344)

[4.9.2 Constraint graph 38](#_Toc152234345)

[4.9.3 Constraint propagation 38](#_Toc152234346)

[4.9.4 Local consistency (node, arc, path) 38](#_Toc152234347)

[4.9.5 AC- 3 Algorithm (Arc consistency ) 39](#_Toc152234348)

[4.9.6 Path consistency 40](#_Toc152234349)

[4.9.7 K-consistency (dạng tổng quát của local consistency) 40](#_Toc152234350)

[4.9.8 Global constraints 41](#_Toc152234351)

[4.9.9 Solving a Sodoku puzzle (AC – 3, Triplets) 41](#_Toc152234352)

[4.9.10 Backtracking search algorithm 41](#_Toc152234353)

[4.9.11 Comments on backtracking search algorithm 43](#_Toc152234354)

[4.9.12 Min-conflict algorithm. 43](#_Toc152234355)

[4.9.13 Tree CSP solver 44](#_Toc152234356)

[5. Game “SOKOBANAI” 46](#_Toc152234357)

[5.1 Mô tả trò chơi 46](#_Toc152234358)

[5.2 Mô tả các class và function 49](#_Toc152234359)

[5.2.1 Class state 49](#_Toc152234360)

[5.2.1.1 Phương thức khởi tạo (\_\_init\_\_): 50](#_Toc152234361)

[5.2.1.2 Phương thức Get Line (get\_line): 51](#_Toc152234362)

[5.2.1.3 Phương thức Tính Heuristic (compute\_heuristic): 51](#_Toc152234363)

[5.2.2 Depth first search 51](#_Toc152234364)

[5.2.3 A\* search 53](#_Toc152234365)

[5.2.4 Các hàm hỗ trợ 56](#_Toc152234366)

[5.2.4.1 Kiểm tra các trạng thái chiến thắng 56](#_Toc152234367)

[5.2.4.2 Tìm vị trí người chơi 56](#_Toc152234368)

[5.2.4.3 Tìm vị trí các hộp 57](#_Toc152234369)

[5.2.4.4 Tìm các vị trí người chơi có thể di chuyển 57](#_Toc152234370)

[5.2.4.5 Hàm di chuyển 59](#_Toc152234371)

[5.2.4.6 Hàm main 60](#_Toc152234372)

[TÀI LIỆU THAM KHẢO 61](#_Toc152234373)

# 1.ĐẠO VĂN

## 1.1 Đạo văn là gì ?

Đạo văn (tiếng Anh: plagiarism) là chiếm hữu một cách sai trái, ăn cắp, công bố ngôn ngữ, suy nghĩ, ý tưởng, hay cách diễn đạt của người khác và xem chúng như là những gì do mình tự tạo ra. Khái niệm đạo văn vẫn chưa có những định nghĩa và quy tắc rõ ràng.Đạo văn được xem là hành vi thiếu trung thực về mặt học thuật và vi phạm đạo đức báo chí. Người nào đạo văn sẽ bị phạt tiền, bị đình chỉ, và thậm chí bị đuổi học hay đuổi việc.Trong môi trường học thuật và công việc, đạo văn là một hành vi vi phạm đạo đức rất nghiêm trọng.

## 1.2 Những điều nên/không nên làm khi làm bài báo cáo

Code cũng là một sản phẩm trí tuệ của người khác. Vì vậy, khi làm báo cáo cần tránh những việc sau đây:

* Sử dụng code của người khác mà không ghi hoặc ghi sai nguồn.
* Không sao chép toàn bộ ý tưởng, code của người khác. Phải biểu diễn kết quả theo cách hiểu của bản thân.
* Tìm hiểu thật sâu và hiểu thật rõ một vấn đề nào đó nào để hạn chế dựa dẫm vào những sản phẩm sẵn có.

## 1.3 Cam kết

Chúng em xin cam đoan đồ án này do các thành viên nhóm thực hiện. Chúng em không sao chép, sử dụng bất kỳ tài liệu, mã nguồn… của người khác mà không ghi nguồn. Chúng em xin chịu hoàn toàn trách nhiệm nếu vi phạm đạo văn

|  |  |
| --- | --- |
| Họ và tên | MSSV |
| Nguyễn Ngọc Mạnh | 21110781 |
| Võ Hoàng Tùng | 21110811 |
| Đặng Văn Thông | 19110056 |
| Nguyễn Ngọc Minh | 21110784 |

# 2. TÌM HIỂU VỀ TRÍ TUỆ NHÂN TẠO

## 2.1 Trí tuệ nhân tạo(Artificial Intelligence) là gì

Trí tuệ nhân tạo hay trí thông minh nhân tạo (AI) là một ngành thuộc lĩnh vực khoa học máy tính. Đây là công nghệ mô phỏng các quá trình suy nghĩ và học tập của con người cho máy móc, đặc biệt là các hệ thống máy tính. Qua đó, trí tuệ nhân tạo giúp máy tính có được những trí tuệ của con người như: Biết suy nghĩ và lập luận để giải quyết vấn đề, biết giao tiếp do hiểu ngôn ngữ, tiếng nói, biết học và tự thích nghi…

Các ứng dụng của AI bao gồm:

* Công cụ Chatbot AI hỗ trợ trả lời câu hỏi
* Công cụ AI hỗ trợ việc viết và chỉnh sửa văn bản
* Công cụ AI giúp tìm kiếm và tạo hình ảnh
* Công cụ AI hỗ trợ tạo âm thanh và video
* Công cụ AI hỗ trợ làm powerpoint, thuyết trình
* Công cụ AI hỗ trợ thiết kế
* Y tế: Chẩn đoán bệnh, nghiên cứu và phát triển thuốc, quản lý dữ liệu sức khỏe
* Giáo dục: Hướng dẫn thông minh, nền tảng kỹ thuật số dùng AI
* Kinh doanh: Tư vấn khách hàng, quản lý dữ liệu, phân tích dữ liệu
* Sản xuất: Tự động hóa, quản lý dữ liệu, phân tích dữ liệu
* Giải trí: Tạo nội dung, tư vấn khách hàng, phân tích dữ liệu

## 2.2 Lích sử hình thành của AI

Lịch sử của Trí tuệ Nhân tạo bắt nguồn từ thời kỳ cổ đại, khi các nhà triết học đầu tiên bắt đầu suy nghĩ về khả năng tồn tại của các sinh vật nhân tạo, máy móc và robot tự động. Những tưởng tượng này đã khơi nguồn cho sự hình thành của trí tuệ nhân tạo, đặc biệt trong thế kỷ 18 khi các nhà triết học bắt đầu nghiên cứu cách máy móc có thể mô phỏng suy nghĩ con người và được điều khiển một cách nhân tạo.

Cuối thế kỷ 18, sự quan tâm đối với AI đã tăng lên, đặc biệt sau khi các nhà triết học, toán học và logic học cổ điển bắt đầu áp dụng các ký hiệu máy móc trong suy nghĩ, dẫn đến sự xuất hiện của máy tính kỹ thuật số đầu tiên như Atanasoff Berry Computer (ABC) vào những năm 1940. Phát minh này đã là nguồn cảm hứng cho sự phát triển của ý tưởng về "bộ não điện tử" hay sinh vật thông minh nhân tạo.

Một bước quan trọng khác là bài kiểm tra Turing do nhà toán học Alan Turing đề xuất vào cuối thập kỷ 1940 để đánh giá khả năng của máy móc trong việc mô phỏng hành động con người một cách khó phân biệt được. Cuối thập kỷ đó, lĩnh vực Trí tuệ Nhân tạo được thiết lập tại một hội nghị tại Đại học Dartmouth vào giữa những năm 1950, nơi thuật ngữ "Trí tuệ Nhân tạo" được đặt ra bởi John McCarthy, nhà khoa học máy tính và nhận thức.

Từ những năm 1950 trở đi, sự tiến bộ trong lĩnh vực này đã được hỗ trợ bởi nhiều nhà khoa học, lập trình viên và nhà lý thuyết khác nhau, làm cho hiểu biết về AI ngày càng phong phú. Mỗi thập kỷ đều chứng kiến những đổi mới và phát hiện mới, làm thay đổi cách chúng ta hiểu về lịch sử và phát triển của AI, từ khái niệm tưởng tượng thành một hiện thực hữu hình cho cả thế hệ hiện tại và tương lai.

## 2.3 Đạo đức trí tuệ nhân tạo

Các nhà nghiên cứu, chuyên gia, và tổ chức liên quan đến trí tuệ nhân tạo đã nỗ lực đưa ra các định nghĩa về AI trong quá trình phát triển, bắt đầu từ thời kỳ hình thành và kéo dài đến sự phát triển đột phá hiện nay của công nghệ này. Một số cách tiếp cận định nghĩa về đạo đức AI có thể mang tính chất tổng quát, không thể hiện đủ tính chất và đặc điểm của công nghệ này. Ngược lại, các cách tiếp cận khác về định nghĩa AI tập trung vào các lĩnh vực phát triển, ứng dụng, hoặc các ảnh hưởng đối với các đối tượng, lĩnh vực mà họ quan tâm. Dưới đây là các định nghĩa cơ bản về đạo đức AI:

* *“Đạo đức AI là tập hợp các giá trị, nguyên tắc, kỹ thuật sử dụng các tiêu chuẩn về đúng, sai được chấp nhận rộng rãi để hướng dẫn các hành vi đạo đức trong phát triển và sử dụng công nghệ AI”* - (Viện Alan Turing).
* *“Đạo đức AI là lĩnh vực nghiên cứu đa nguyên tắc và nhiều bên liên quan nhằm xác định và thực thi các giải pháp kỹ thuật cũng như phi kỹ thuật để giải quyết các mối quan tâm và giảm thiểu các rủi ro mà công nghệ này mang lại”* - (Diễn đàn Kinh tế thế giới).
* *Đạo đức AI là “Sự phản ánh có tính hệ thống và tính quy phạm một cách toàn diện, tổng thể và theo khung phát triển các giá trị, nguyên tắc và hành động phụ thuộc lẫn nhau nhằm hướng dẫn cộng đồng, xã hội đối phó một cách có trách nhiệm với các tác động đã biết và chưa biết của công nghệ AI đối với con người, xã hội, môi trường và hệ sinh thái; đồng thời cung cấp cho họ cơ sở để chấp nhận hoặc từ chối các công nghệ AI”* - (UNESCO).

Có nhiều phương hướng tiếp cận về đạo đức của AI. Có 3 hướng tiếp cận cơ bản:

* Virtue Ethics (Đạo đức học đức hạnh):

Đạo đức đức hạnh là một triết lý được phát triển bởi Aristotle và những người Hy Lạp cổ đại khác. Đó là sự tìm hiểu và sống một cuộc sống có đạo đức.

Cách tiếp cận đạo đức dựa trên tính cách này giả định rằng chúng ta có được đức tính thông qua thực hành. Bằng cách rèn luyện tính trung thực, dũng cảm, công bằng, rộng lượng, ..., một người sẽ phát triển một nhân cách đáng kính và có đạo đức. Theo Aristotle, bằng cách rèn giũa những thói quen đạo đức, con người sẽ có thể đưa ra lựa chọn đúng đắn khi đối mặt với những thách thức về đạo đức. Đạo đức học đức hạnh giúp chúng ta hiểu được ý nghĩa của việc làm người có đạo đức. Và, nó cho chúng ta một hướng dẫn để sống cuộc sống mà không đưa ra cho chúng ta những quy tắc cụ thể để giải quyết tình thế tiến thoái lưỡng nan về mặt đạo đức.

Nhưng khi đến giai đoạn công nghiệp phát triển, tính sở hữu được đề cao và các vấn đề cũng bắt đầu được hình thành. Đạo đức học đức hạnh đã không đủ khả năng kiểm soát, lúc này một trường phái đạo đức mới ra đời

* Deontology (Đạo đức sinh học):

Đạo đức sinh học là một lý thuyết đạo đức sử dụng các quy tắc để phân biệt đúng sai. Nghĩa vụ học thường gắn liền với triết gia Immanuel Kant. Kant tin rằng các hành động đạo đức tuân theo các quy luật đạo đức phổ quát, chẳng hạn như “Đừng nói dối. Đừng ăn trộm. Đừng lừa dối.”

Đạo đức sinh học rất đơn giản để áp dụng. Nó chỉ yêu cầu mọi người tuân theo các quy tắc và thực hiện nghĩa vụ của mình. Cách tiếp cận này có xu hướng phù hợp với trực giác tự nhiên của chúng ta về điều gì là đạo đức hoặc điều gì không đạo đức.

Deotology đã phổ biến và được nhiều người sử dụng. Tuy nhiên, nó khá cứng nhắc và đã có tình huống không được xử lý công bằng. Vì thế một trường phái khác được ra đời để khắc phục những sai số đó.

* Consequentialism (Chủ nghĩa hệ quả):

Chủ nghĩa hệ quả là một lý thuyết đạo đức đánh giá liệu điều gì đó có đúng hay không thông qua hậu quả của nó. Ví dụ, hầu hết mọi người đều đồng ý rằng nói dối là sai. Nhưng nếu nói dối có thể giúp cứu mạng một người thì chủ nghĩa hệ quả cho rằng đó là điều đúng đắn nên làm.

Chủ nghĩa hệ quả đôi khi bị chỉ trích vì khó hoặc thậm chí không thể biết trước kết quả của một hành động sẽ như thế nào. Quả thực, không ai có thể biết chắc chắn về tương lai. Ngoài ra, trong một số tình huống nhất định, chủ nghĩa hệ quả có thể dẫn đến những quyết định có thể bị phản đối, mặc dù hậu quả được cho là tốt.

## 2.4 Cách tiếp cận của AI

***Thinking humanly***: thường được áp dụng trong lĩnh vực thần kinh. Thành tựu tiêu biểu: mạng nơ-ron Neural Network nhân tạo (thành công nhất trong lĩnh vực AI)

***Thinking rationally:*** dựa theo quy tắc logic mà không nhất thiết phải làm theo hành động của con người.

***Acting humanly***: hành động giống con người mặc kệ thuật toán như thế nào. (ví dụ: Turing Test).

***Acting rationally:*** tìm ra giải pháp (solution) của một vấn đề bằng một chuỗi hành động (actions).

# 3. CÁC THUẬT TOÁN TÌM KIẾM CƠ BẢN

## 3.1 Thuật toán tìm kiếm là gì ?

Thuật toán tìm kiếm là các thuật toán nhằm để giải quyết các vấn đề cần tìm kiếm.

Một thuật toán tìm kiếm bao gồm search space (không gian tìm kiếm), initial state (trạng thái bắt đầu) và goal state (trạng thái đích). Các thuật toán tìm kiếm giúp các AI agents (tác nhân AI) đạt được goal state thông qua việc đánh giá các tình huống và phương án khác nhau.

Các thuật toán cung cấp các phương án tìm kiếm thông qua một các possible actions để đưa agent từ initial state sang goal state. Nếu thiếu các thuật toán tìm kiếm này, các chương trình hay máy móc AI không thể thực hiện các chức năng tìm kiếm và tìm ra được giải pháp khả thi.

## 3.2 Các bước cơ bản của một thuật toán tìm kiếm

Bước 1: Chọn một node để mở rộng.

Bước 2: Mở rộng node được chọn và tìm các possible actions.

Lặp lại 2 bước trên cho đến khi tìm được trạng thái đích hoặc không còn node nào để mở rộng nữa. Tất cả các thuật toán tìm kiếm đều sẽ bao gồm 2 bước trên, chỉ khác ở cách chọn node ở bước 1.

## 3.3 Tiêu chí đánh giá thuật toán

Có thể đánh giá một thuật toán dựa vào 4 tiêu chí cơ bản:

* Completeness (Tính hoàn chỉnh): một thuật toán được coi là hoàn chỉnh khi cung cấp cho nó một đầu vào và có ít nhất một đầu ra cho bài toán.
* Optimality (Tính tối ưu): Các thuật toán tìm kiếm còn được đặc trưng bởi các lời giải tối ưu. Tối ưu ở đây được hiểu là thuật toán có thể tìm được đường đi tốt nhất với một chi phí thấp nhất.
* Time complexity (Độ phức tạp về thời gian): là thời gian cần thiết để một thuật toán hoàn thành một nhiệm vụ. Độ phức tạp này thường phụ thuộc vào kích thước của đầu vào của thuật toán.
* Space complexity (Độ phức tạp về không gian): là lượng bộ nhớ cần thiết để lưu trữ dữ liệu trong quá trình thực thi của một thuật toán. Độ phức tạp này cũng thường phụ thuộc vào kích thước của đầu vào của thuật toán.

## 3.4 Các thành phần của một vấn đề

Trước khi giải quyết một vấn đề, chúng ta cần phải phân tích một vấn đề ra thành 5 thành phần. Việc định nghĩa các yếu tố này cung cấp cơ sở cho việc tìm kiếm và cung cấp giải pháp để cho vấn đề cần giải quyết. Dưới đây là các yếu tố cần phải xác định:

* Intial state (Trạng thái ban đầu): Điểm bắt đầu của thuật toán
* Possible actions (Hành động khả thi): Đây là các hành động mà agent có thể được áp dụng ở mỗi state nhất định
* Transition model (Mô hình chuyển tiếp): Là kết quả của một action tại một state
* Goal test (Kiểm tra đích): Kiểm tra để xác định xem một state cụ thể có phải là goal state hay không.
* Cost (Chi phí): Thước đo hiệu suất.

## 3.5 Conceptual algorithm (Thuật toán mẫu )

### 3.5.1 Conceptual algorithm #1 – TREE SEARCHA text on a white background Description automatically generated

Một số thuật ngữ:

* Solution: Đường đi từ vị trí bắt đầu đến vị trí cần tìm.
* Frontier: Một tập hợp chứa những node mà thuật toán có thể expand ra.
* Goal state: Trạng thái đích. Nhiệm vụ của thuật toán là tìm được goal state này.
* Expand: Từ một node được chọn thì có thể mở rộng ra được các node khác có thể đi đến được từ node chọn.

Mô tả thuật toán:

* Input: problem (5 components: Initial state, possible actions, transition model, goal test, cost)
* Output: solution hoặc là failure

Cách thuật toán hoạt động:

* Khởi tạo tập frontier chứa initial state của problem
* Thực hiện vòng lặp và kiểm tra:
* Nếu như tập frontier rỗng nhưng vẫn chưa tìm được solution thì trả về failure và ngược lại thì ta lấy một node lá từ trong tập hợp và loại node này ra khỏi tập hợp.
* Kiểm tra nếu node được chọn chứa goal state thì sẽ trả về solution, ngược lại node được chọn không phải là goal state thì hàm sẽ expand node được chọn và thêm những node mới vào trong frontier.
* Thực hiện vòng lặp đến khi nào tìm được goal state hoặc khi tập hợp frontier rỗng.

Chú ý: thuật toán TREE-SEARCH sẽ có trường hợp lặp lại những node đã đi qua và hàm sẽ bị lặp đến vô hạn. Để giải quyết vấn đề này thì ta sẽ tiếp tục đến với Conceptual algorithm #2 GRAPH SEARCH.

### 3.5.2 Conceptual algorithm #2 GRAPH SEARCH

Tương tự như TREE SEARCH nhưng sẽ có thêm một tập explored set chứa các node đã được mở rộng giúp tránh được lặp vô tận.

A white text with black text

Description automatically generatedPseudocode:

Mô tả thuật toán:

* Input: problem (5 components: Initial state, possible actions, transition model, goal test, cost)
* Output: solution hoặc là failure

Cách thuật toán hoạt động:

* Khởi tạo tập frontier chứa initial state của problem
* Khởi tạo tập explored dùng để chứa các node đã được mở rộng
* Thực hiện vòng lặp và kiểm tra:
* Nếu như tập frontier rỗng nhưng vẫn chưa tìm được solution thì trả về failure và ngược lại thì ta lấy một node lá từ trong tập hợp và loại node này ra khỏi tập hợp.
* Kiểm tra nếu node được chọn chứa goal state thì sẽ trả về solution. Nếu chưa thì sẽ thêm node này vào trong tập explored.
* Expand node được chọn và thêm những node mới vào trong frontier khi những node mới này chưa có trong tập frontier hay explored.
* Thực hiện cho đến khi tìm được goal state.

## 3.6 Phân loại thuật toán tìm kiếm

Thuật toán tìm kiếm được chia ra thành 2 loại:

* Uninformed search (Tìm kiếm mù):
  + Không được cung cấp bất kì thông tin nào ngoài các thông tin được cung cấp trong định nghĩa của một problem (5 components).
  + Chọn node một cách ngẫu nhiên để expand.
* Informed search (Tìm kiếm có thông tin):
  + Ngoài 5 components được biết từ problem thì informed search sẽ được cung cấp thêm những gợi ý hay thông tin giúp hỗ trợ giải quyết vấn đề.
  + Dựa vào thông tin được cung cấp thì ta sẽ có một ước lượng chi phí cho node n là f(n), từ đó thuật toán sẽ chọn node có f(n) nhỏ nhất để expand.

## 3.7 Cấu trúc dữ liệu

### 3.7.1 Thành phần của một node

n.State: state tại node đang đại diện.

n.Parent: là node mà node được chọn được expand ra.

n.Action: là hành động mà node cha thực hiện để đến được node được chọn.

n.Path-cost: là chi phí từ initial state cho đến node đang được xét.

CHILD-NODE(problem, parent, action): hàm sinh ra node con từ node cha và action tương ứng.

### 3.7.2 Queue (hàng đợi)

* Toán tử của queue:
  + Empty?(queue): hàm check xem trong queue còn phần tử hay không.
  + Pop(queue): hàm lấy dữ liệu từ một điểm được quy định trong queue.
  + Insert(element, queue): hàm thêm dữ liệu vào trong queue.
* 3 loại queue cơ bản:
  + FIFO queue (first-in, first-out): phần tử vào trước thì sẽ được lấy ra trước.
  + LIFO queue (last-in, first-out **aka** a stack): phần tử đứng vị trí cuối cùng trong queue thì sẽ được lấy ra trước.
  + Priority queue: phần tử được lấy ra sẽ tương ứng với độ ưu tiên của phần tử.

## 3.8 Uninformed search algorithm

### 3.8.1 Breadth first search (BFS)

A screenshot of a computer program

Description automatically generatedPseudocode

Mô tả thuật toán:

* Input: problem (5 components: Initial state, possible actions, transition model, goal test, cost)
* Output: solution hoặc là failure

Chiến lược của thuật toán:

* Chọn node gốc làm initial state và expand đầu tiên.
* Những node được expand từ node gốc sẽ được expand tiếp.
* Nếu như tất cả các node con từ gốc được expand xong thì mới tiếp tục expand tiếp các node có độ sâu lớn hơn.

Cách thuật toán hoạt động:

* Kiểm tra xem state của node được chọn có phải là goal state hay không. Nếu là goal state thì trả về solution.
* Khởi tạo tập frontier sử dụng FIFO queue và thêm node của initial state.
* Khởi tạo tập explored dùng để chứa các node đã được expand.
* Thực hiện vòng lặp:
  + Nếu tập frontier rỗng thì sẽ trả về failure, ngược lại sẽ lấy 1 node từ frontier và thêm state của node này vào tập explored.
  + Thực hiện thêm vòng lặp dùng để expand node được chọn từ frontier. Với mỗi action của node được chọn thì sẽ sinh ra các node con. Nếu state của node con không tồn tại trong explored hay frontier thì xét xem state của node con có phải là goal state hay không. Nếu là goal state thì trả về solution, nếu không thì thêm node con vào frontier.
  + Tiếp tục thực hiện vòng lặp.

Đánh giá thuật toán:

* Completeness: luôn cho ra được solution nếu có tồn tại.
* Optimality: nếu prolem thỏa điều kiên là PATH-COST tăng dần theo độ sâu thì thuật toán BFS sẽ là optimal solution.
* Complexity: thuật toán sẽ đi qua toàn bộ search space nên rất tốn kém thời gian và bộ nhớ.

### 3.8.2 Uniform cost search

Pseudocode:A screenshot of a computer program

Description automatically generated

Chiến lược thuật toán: tương tự như BFS.

Cách thuật toán hoạt động:

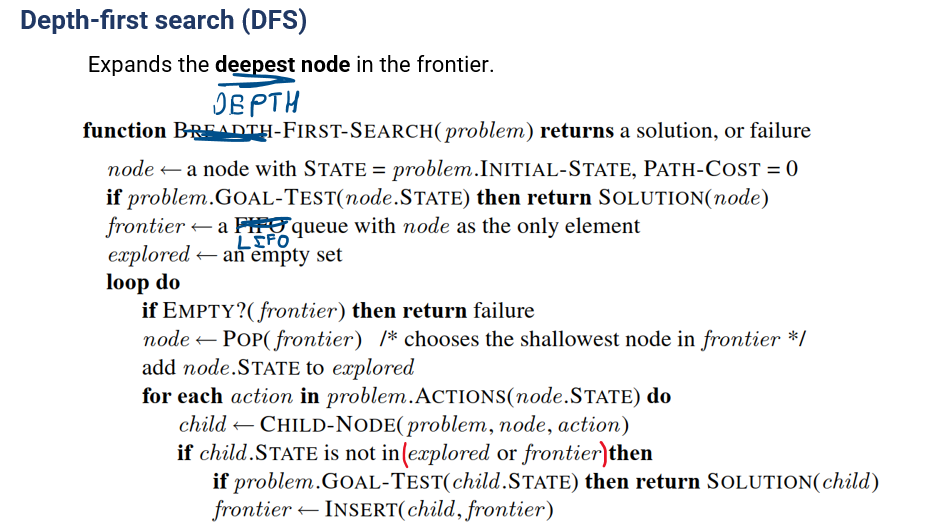
* Kiểm tra xem state của node được chọn có phải là goal state hay không. Nếu là goal state thì trả về solution.
* Khởi tạo tập frontier sử dụng Priority queue được sắp xếp theo PATH-COST và thêm node của initial state.
* Khởi tạo tập explored dùng để chứa các node đã được expand.
* Thực hiện vòng lặp:
  + Nếu tập frontier rỗng thì sẽ trả về failure, ngược lại sẽ lấy 1 node từ frontier. Khác với BFS thì Uniform-cost search sẽ lấy ra node có cost thấp nhất trong frontier để xét và thêm state của node này vào tập explored.
  + Thực hiện thêm vòng lặp dùng để expand node được chọn từ frontier. Với mỗi action của node được chọn thì sẽ sinh ra các node con. Nếu state của node con không tồn tại trong explored hay frontier thì xét xem state của node con có phải là goal state hay không. Nếu là goal state thì trả về solution, nếu không thì thêm node con vào frontier. Còn nếu node con đã tồn tại trong frontier thì thuật toán sẽ so sánh nếu node con đã tồn tại có PATH-COST cao hơn thì sẽ bị thay thế bởi node con đang được xét.
  + Tiếp tục thực hiện vòng lặp.

Đánh giá thuật toán:

* Completeness: luôn cho ra được solution nếu có tồn tại.
* Optimality: luôn luôn đưa ra được optimal solution
* Complexity: thuật toán sẽ đi qua toàn bộ search space nên rất tốn kém thời gian và bộ nhớ.

### 3.8.3 Depth first search(DFS)

Pseudocode:



Cách thuật toán hoạt động:

* Gần như là tương tự với BFS và chỉ khác là tập frontier của DFS sẽ sử dụng LIFO queue. Chính vì sử dụng LIFO nên thuật toán sẽ duyệt theo chiều sâu, nó sẽ duyệt một nhánh và duyệt hết nhánh này rồi loại bỏ nhánh này hoàn toàn ra khỏi bộ nhớ rồi mới duyệt đến nhánh khác cho đến khi tìm được goal state thì sẽ trả về solution. Còn nếu không tìm được thì sẽ trả về failure.

Đánh giá thuật toán:

* Completeness: có khả năng thuật toán sẽ duyệt trúng vào một nhánh có độ dài vô hạn và không tìm được goal state.
* Optimality: không đưa ra được optimal solution
* Complexity: bởi vì khi duyệt qua một nhánh mà không có goal state thì thuật toán sẽ xóa toàn bộ nhánh ra khỏi bộ nhớ nên sẽ tiết kiệm được bộ nhớ.

### 3.8.4 Depth limited search (DLS)

Để giải quyết vấn đề của thuật toán DFS là Completeness thì ta sẽ sử dụng thuật toán DLS bằng cách ta sẽ ước lượng và giới hạn lại chiều sâu của nhánh được duyệt.

A screenshot of a computer program

Description automatically generatedPseudocode:

Mô tả thuật toán:

* Input: problem, limit (chiều sâu giới hạn mà thuật toán có thể tìm kiếm).
* Output: solution hoặc failure/cutoff (khi đến chiều sâu giới hạn mà không tìm được goal state).
* Sử dụng đệ quy.

Chiến lược thuật toán:

* Chương trình sẽ gọi hàm đệ quy nhận vào các tham số là state hiện tại và limit. Và sẽ duyệt tiếp xuống theo chiều sâu cho đến khi tìm được goal state hoặc tìm tới giới hạn độ sâu mà vẫn không tìm ra được solution thì trả về failure hoặc cutoff.

Cách thuật toán hoạt động:

* Đệ quy hàm RECURSIVE\_DLF, hàm này sẽ nhận vào node, problem, limit và trả về solution hoặc failure/cutoff.
* Kiểm tra xem state hiện tại có phải là goal state hay không. Nếu là goal state thì trả về solution hoặc nếu limit bằng 0 tức là đã duyệt đến độ sâu được quy định thì trả về cutoff.
* Nếu 2 điều kiện trên chưa thỏa thì gán cutoff\_occurred là false.
* Thực hiện vòng lặp qua từng action để expand các node con. Gọi đệ quy hàm RECURSIVE\_DLF và truyền vào node con vừa được expand, problem, limit – 1.
* Thực hiện đệ quy đến khi hàm trả về cutoff thì gán cutoff\_occurred là true. Nếu hàm trả về là solution thì kết thúc đệ quy.
* Kết thúc vòng lặp và kiểm tra biến cutoff\_occured nếu là false thì trả về failure còn true thì trả về cutoff.

Nhưng DLS gặp một vấn đề là làm thế nào để ước lượng được độ sâu như thế nào là hợp lí. Vậy nên chúng ta cần một thuật toán khác tối ưu hơn.

### 3.8.5 Iterative deepening search

Đây là thuật toán tối ưu nhất trong các thuật toán Uninformed search. Nó sẽ giúp giải quyết được các vấn đề của các thuật toán trên.

A close up of words

Description automatically generatedPseudocode:

Cách hoạt động của thuật toán:

* Tạo một vòng lặp với độ sâu chạy từ 0 cho đến vô hạn và chạy thuật toán DLS cho đến khi nào tìm được kết quả thì trả về solution còn không thì trả về failure.

## 3.9 Informed search algorithm

### 3.9.1 Các hàm ước lượng quãng đường

Trong các thuật toán informed search như A\* và Best-First Search, hàm ước lượng quãng đường (heuristic function) được sử dụng để ước tính chi phí đi từ trạng thái hiện tại đến trạng thái mục tiêu. Hàm này có thể sử dụng để hướng dẫn quá trình tìm kiếm, giúp thuật toán tìm thấy lời giải nhanh hơn.

* Hàm heuristic Euclidean (Euclidean heuristic): hàm tính khoảng cách Euclidean (đường thẳng) từ vị trí hiện tại đến vị trí đích. Đây là hàm ước lượng tốt nhưng có thể dẫn đến việc tìm kiếm không tối ưu trong một số trường hợp.
* Hàm heuristic Manhattan (Manhattan heuristic): hàm tính khoảng cách Manhattan (tổng các đoạn đường theo hướng ngang và dọc) từ vị trí hiện tại đến vị trí đích. Hàm ước lượng này đảm bảo tìm kiếm tối ưu trong trường hợp các nút chỉ di chuyển theo các hướng ngang và dọc.
* Hàm heuristic Chebyshev (Chebyshev heuristic): hàm tính khoảng cách Chebyshev (khoảng cách theo hình vuông) từ vị trí hiện tại đến vị trí đích. Hàm ước lượng này thường được sử dụng trong các lưới vuông và cho phép di chuyển theo các hướng ngang, dọc và chéo.
* Hàm heuristic Octile (Octile heuristic): hàm tính khoảng cách Octile (khoảng cách Octile giữa các ô trong lưới) từ vị trí hiện tại đến vị trí đích. Hàm ước lượng này cũng cho phép di chuyển theo các hướng ngang, dọc và chéo.

Các hàm heuristic trên được sử dụng để tính giá trị h(n) trong công thức:

**f(n) = g(n) + h(n)**

Trong đó:

* g(n) là độ dài đường đi từ nút gốc tới nút hiện tại
* h(n) là ước lượng khoảng cách từ nút hiện tại đến nút đích
* f(n) là giá trị ước lượng tổng cộng.

Lựa chọn hàm heuristic phù hợp sẽ ảnh hưởng đến hiệu suất và tính tối ưu của thuật toán informed search. Tùy thuộc vào bài toán cụ thể và đặc điểm của môi trường, có thể áp dụng một hàm heuristic thích hợp để đạt được kết quả tốt nhất.

### 3.9.2 Ý tưởng chung của informed search

Ý tưởng chung của thuật toán informed search (thuật toán tìm kiếm có thông tin) là sử dụng đích đến để hướng dẫn quá trình tìm kiếm. Trong các thuật toán informed search, thông tin này được thể hiện bằng cách sử dụng một hàm heuristic để đánh giá quãng đường còn lại từ một nút đến đích.

Các thuật toán informed search sử dụng thông tin heuristic để ước lượng giá trị tốt nhất của các nút chưa được khám phá. Thuật toán sẽ ưu tiên khám phá các nút có giá trị heuristic thấp hơn, coi chúng là tiềm năng tốt nhất để đạt được đích.

Các bước chính của một bài toán informed search:

* Khởi tạo nút gốc và nút đích.
* Khởi tạo danh sách open (danh sách các nút chưa được khám phá) và danh sách closed (danh sách các nút đã được khám phá).
* Đặt giá trị heuristic cho nút gốc.
* Lặp lại cho đến khi tìm thấy đường đi hoặc không còn nút nào trên danh sách open:
* Chọn nút có giá trị f = g + h nhỏ nhất từ danh sách open.
* Nếu nút này là nút đích, đã tìm thấy đường đi.
* Di chuyển nút này từ danh sách open sang danh sách closed.
* Kiểm tra và mở rộng các nút con của nút hiện tại (các nút kề và chưa được khám phá).
* Cập nhật giá trị f, g và h cho các nút con và thêm chúng vào danh sách open.
* Nếu đã tìm thấy đường đi, truy vết từ nút đích về nút gốc để tạo ra đường đi tối ưu.

Các thuật toán informed search sử dụng thông tin heuristic để tối ưu hóa quá trình tìm kiếm và giảm độ phức tạp thời gian. Bằng cách ước lượng giá trị tốt nhất từ nút hiện tại đến đích, thuật toán có khả năng tìm kiếm và khám phá các đường đi tiềm năng tốt nhất trước, giúp giảm số lượng nút phải khám phá và tăng tốc quá trình tìm kiếm.

### 3.9.3 Best first search

Thuật toán Best-First Search là một thuật toán tìm kiếm có thông tin (informed search) dựa trên nguyên tắc chọn nút tiềm năng tốt nhất để khám phá tiếp theo. Thuật toán này không đảm bảo tìm ra đường đi tối ưu, nhưng có thể tìm được một đường đi nhanh chóng trong một số trường hợp.

Dưới đây là mô tả cách hoạt động của thuật toán Best-First Search:

* + Khởi tạo nút gốc và nút đích.
  + Khởi tạo danh sách open (danh sách các nút chưa được khám phá) và danh sách closed (danh sách các nút đã được khám phá).
  + Đặt giá trị heuristic cho nút gốc.
  + Thêm nút gốc vào danh sách open.
  + Lặp lại cho đến khi tìm thấy đường đi hoặc không còn nút nào trên danh sách open:
* Chọn nút có giá trị heuristic tốt nhất từ danh sách open.
* Nếu nút này là nút đích, đã tìm thấy đường đi.
* Di chuyển nút này từ danh sách open sang danh sách closed.
* Kiểm tra và mở rộng các nút con của nút hiện tại (các nút kề và chưa được khám phá).
* Cập nhật giá trị heuristic cho các nút con và thêm chúng vào danh sách open.
  + Nếu đã tìm thấy đường đi, truy vết từ nút đích về nút gốc để tạo ra đường đi.

Thuật toán Best-First Search dựa vào giá trị heuristic để quyết định nút nào sẽ được khám phá tiếp theo. Giá trị heuristic được dùng để ước lượng giá trị tốt nhất từ nút hiện tại đến nút đích. Thuật toán tìm kiếm các nút có giá trị heuristic thấp hơn, coi chúng là tiềm năng tốt nhất để đạt được đích. Điều này có thể dẫn đến việc tìm ra một đường đi nhanh chóng, nhưng không đảm bảo tối ưu.

Best-First Search không xem xét chi phí di chuyển từ nút gốc đến nút hiện tại, chỉ dựa vào giá trị heuristic. Do đó, nếu không có một hàm heuristic tốt, thuật toán có thể dẫn đến việc khám phá các đường đi không khả thi hoặc không tối ưu.

### 3.9.4 A\* search

Thuật toán A\* Search là một thuật toán tìm kiếm có thông tin (informed search) kết hợp cả yếu tố tìm kiếm theo chiều rộng (breadth-first search) và yếu tố tìm kiếm theo chiều sâu (depth-first search). Thuật toán này sử dụng hàm heuristic để ước lượng giá trị tốt nhất từ nút hiện tại đến nút đích và đảm bảo tìm được đường đi tối ưu.

Dưới đây là mô tả cách hoạt động của thuật toán A\* Search:

* + Khởi tạo nút gốc và nút đích.
  + Khởi tạo danh sách open (danh sách các nút chưa được khám phá) và danh sách closed (danh sách các nút đã được khám phá).
  + Đặt giá trị g(n) của nút gốc bằng 0 (độ dài đường đi từ nút gốc đến nút hiện tại).
  + Đặt giá trị heuristic h(n) cho nút gốc (ước lượng giá trị tốt nhất từ nút hiện tại đến nút đích).
  + Tính giá trị f(n) = g(n) + h(n) cho nút gốc và thêm nút này vào danh sách open.
  + Lặp lại cho đến khi tìm thấy đường đi hoặc không còn nút nào trên danh sách open:
* Chọn nút có giá trị f(n) nhỏ nhất từ danh sách open.
* Nếu nút này là nút đích, đã tìm thấy đường đi.
* Di chuyển nút này từ danh sách open sang danh sách closed.
* Kiểm tra và mở rộng các nút con của nút hiện tại (các nút kề và chưa được khám phá).
* Cập nhật giá trị g(n) và h(n) cho các nút con.
* Tính giá trị f(n) = g(n) + h(n) cho các nút con và thêm chúng vào danh sách open.
* Nếu đã tìm thấy đường đi, truy vết từ nút đích về nút gốc để tạo ra đường đi tối ưu.

Thuật toán A\* Search sử dụng giá trị f(n) = g(n) + h(n) để ước lượng giá trị tốt nhất từ nút hiện tại đến nút đích. Giá trị g(n) đại diện cho độ dài đường đi từ nút gốc đến nút hiện tại, trong khi giá trị h(n) là giá trị heuristic ước lượng từ nút hiện tại đến nút đích. Thuật toán ưu tiên khám phá các nút có giá trị f(n) thấp hơn, coi chúng là tiềm năng tốt nhất để đạt được đích.

Với việc sử dụng cả yếu tố tìm kiếm theo chiều rộng và chiều sâu, A\* Search có khả năng tìm kiếm một đường đi tối ưu trong một thời gian hợp lý. Tuy nhiên, hiệu suất của thuật toán phụ thuộc vào chất lượng của hàm heuristic được sử dụng. Một hàm heuristic tốt sẽ giúp A\* Search tìm ra đường đi tối ưu nhanh chóng, trong khi một hàm heuristic kém có thể dẫn đến tìm kiếm không tối ưu hoặc chậm chạp.

### 3.9.5 Cách tạo ra heuristic funcion

Khi tạo heuristic function cho bài toán informed search, bạn cần xem xét bài toán cụ thể và các thông tin có sẵn để đưa ra một ước lượng tốt về giá trị từ nút hiện tại đến nút đích. Dưới đây là một số hướng dẫn để tạo heuristic function cho bài toán informed search:

* + Phân tích bài toán: Hiểu rõ bài toán và các ràng buộc của nó. Điều này đòi hỏi bạn nắm vững về các yếu tố quan trọng trong bài toán và cách chúng ảnh hưởng đến quyết định tìm kiếm.
  + Xem xét thông tin đã biết: Xem xét thông tin có sẵn trong bài toán. Điều này có thể bao gồm các chi phí di chuyển, thông tin về môi trường, điều kiện, hoặc bất kỳ điều gì có thể giúp định hướng tìm kiếm đến mục tiêu.
  + Định nghĩa các trạng thái và hành động: Xác định các trạng thái và hành động trong bài toán. Điều này giúp bạn hiểu được cách mà các yếu tố khác nhau có thể ảnh hưởng đến quyết định tìm kiếm và định hướng của heuristic function.
  + Xác định các yếu tố quan trọng: Xác định các yếu tố quan trọng trong bài toán. Điều này có thể là khoảng cách, thời gian, chi phí hoặc bất kỳ yếu tố nào khác mà bạn muốn tối ưu hoá trong quá trình tìm kiếm.
  + Tạo heuristic function: Dựa trên thông tin đã thu thập được và các yếu tố quan trọng, bạn có thể tạo heuristic function. Heuristic function nên đưa ra một ước lượng tốt về giá trị từ nút hiện tại đến nút đích. Điều quan trọng là heuristic function phải hợp lý và không quá lạc quan hoặc quá bi quan.
  + Kiểm tra và hiệu chỉnh: Kiểm tra và hiệu chỉnh heuristic function của bạn. Đánh giá tính hiệu quả và độ chính xác của heuristic function bằng cách áp dụng nó vào thuật toán informed search và xem kết quả tìm kiếm có tối ưu không. Nếu kết quả không tốt, bạn có thể điều chỉnh heuristic function để cải thiện hiệu suất.

### 3.9.6 So sánh hàm heuristic

Các hàm heuristic có thể được so sánh dựa trên hai tiêu chí chính: ước lượng tốt và tính khả thi.

* + Ước lượng tốt: Một hàm heuristic được coi là tốt nếu nó đưa ra ước lượng gần đúng về giá trị thực tế từ nút hiện tại đến nút đích. Một hàm heuristic tốt sẽ giúp thuật toán informed search tìm đường đi tối ưu nhanh chóng và hiệu quả hơn. Nếu một hàm heuristic không đưa ra ước lượng tốt, thuật toán có thể mất thời gian tìm kiếm nhiều hướng không cần thiết hoặc không tìm được đường đi tối ưu.
  + Tính khả thi: Một hàm heuristic được coi là khả thi nếu nó có thể tính toán một cách hiệu quả và không tốn nhiều thời gian. Tính khả thi của hàm heuristic là quan trọng để đảm bảo rằng việc tính toán ước lượng không làm chậm quá trình tìm kiếm. Nếu một hàm heuristic quá phức tạp hoặc tốn nhiều thời gian để tính toán, nó có thể làm giảm hiệu suất của thuật toán informed search.

Các hàm heuristic cũng có thể được so sánh dựa trên hiệu suất và chính xác trong các bài toán cụ thể. Một hàm heuristic có thể hoạt động tốt trong một bài toán nhất định nhưng không hiệu quả trong bài toán khác. Do đó, việc lựa chọn hàm heuristic phụ thuộc vào từng bài toán cụ thể và kiến thức về vấn đề đang được giải quyết.

Ngoài ra, cách so sánh các hàm heuristic còn phụ thuộc vào mục tiêu của bài toán. Một hàm heuristic có thể tốt trong việc tìm kiếm đường đi ngắn nhất, trong khi hàm heuristic khác có thể tốt trong việc tìm kiếm đường đi đáng tin cậy hoặc tối ưu theo mục tiêu khác.

Vì vậy, không có một hàm heuristic duy nhất phù hợp cho tất cả các bài toán. Việc lựa chọn và so sánh các hàm heuristic phụ thuộc vào bài toán cụ thể và yêu cầu của nó.

# 4. LOCAL SEARCH – SEARCHING FOR GOAL STATE

## 4.1 State space lanscape

Landscape của không gian trạng thái (state space) là một khái niệm được sử dụng để mô tả cấu trúc và đặc điểm của các trạng thái trong một bài toán cụ thể. Nó thể hiện sự biến đổi của giá trị hàm mục tiêu hoặc hàm đánh giá trong không gian trạng thái và có thể được biểu diễn dưới dạng biểu đồ, biểu đồ đường cong, hoặc các biểu đồ khác.

Landscape có thể có các đặc điểm và đường dẫn khác nhau, phản ánh sự biến thiên của hàm mục tiêu hoặc hàm đánh giá trong không gian trạng thái.

* + Đỉnh (Peak): Đỉnh trong landscape đại diện cho các trạng thái tốt nhất trong không gian trạng thái. Đó là các trạng thái có giá trị hàm mục tiêu hoặc hàm đánh giá cao nhất. Một landscape có nhiều đỉnh có thể cho thấy một bài toán có nhiều giải pháp tốt.
  + Thung lũng (Valley): Thung lũng trong landscape là các vùng thấp nằm giữa các đỉnh. Đó là các trạng thái có giá trị hàm mục tiêu hoặc hàm đánh giá thấp. Thung lũng có thể đại diện cho các giải pháp cục bộ tối ưu hơn mà thuật toán có thể mắc kẹt.
  + Đường dốc (Slope): Đường dốc trong landscape biểu thị sự thay đổi của giá trị hàm mục tiêu hoặc hàm đánh giá khi di chuyển từ một trạng thái sang trạng thái khác. Đường dốc dốc tăng biểu thị sự cải thiện của giá trị, trong khi đường dốc dốc giảm biểu thị sự giảm giá trị.
  + Điểm yên ngựa (Saddle Point): Điểm yên ngựa là các điểm trong landscape nằm giữa các thung lũng và có đường dốc xung quanh không cho phép tiến bước tiếp theo. Điểm yên ngựa có thể gây khó khăn cho thuật toán tìm kiếm trong việc thoát khỏi các giải pháp cục bộ.
  + Đường cong và cạnh: Trong landscape, đường cong và cạnh biểu thị các quan hệ và sự chuyển đổi giữa các trạng thái trong không gian trạng thái. Các cạnh nối các trạng thái và đường cong biểu diễn sự biến đổi của giá trị hàm mục tiêu hoặc hàm đánh giá khi di chuyển từ trạng thái này sang trạng thái khác.

Một số thuật toán phổ biến trong việc tìm kiếm trong không gian trạng thái bao gồm:

* + Thuật toán Hill Climbing: Thuật toán Hill Climbing tìm kiếm và di chuyển lên các trạng thái có giá trị hàm mục tiêu tốt hơn. Nó duyệt qua các trạng thái kề cận và chọn trạng thái tốt nhất để di chuyển đến.
  + Thuật toán Simulated Annealing: Thuật toán Simulated Annealing kết hợp việc khám phá các khu vực mới và tránh rơi vào các giải pháp cục bộ tốt nhất. Nó sử dụng một tham số gọi là "nhiệt độ" để điều chỉnh mức độ khám phá và khả năng chấp nhận các giải pháp kém hơn.

## 4.2 Local search

Local search (tìm kiếm cục bộ) là một thuật toán tìm kiếm trong lĩnh vực trí tuệ nhân tạo và tối ưu hóa. Nó tập trung vào việc khám phá và cải thiện các giải pháp trong một vùng xung quanh của trạng thái hiện tại, thay vì tìm kiếm trên toàn bộ không gian trạng thái.

Thuật toán tìm kiếm cục bộ bắt đầu bằng cách tạo ra một trạng thái ban đầu hoặc tìm kiếm trong không gian trạng thái để tìm một trạng thái khả thi. Sau đó, nó sử dụng các quy tắc di chuyển để tạo ra các trạng thái kề cận và đánh giá chất lượng của chúng bằng hàm mục tiêu hoặc hàm đánh giá. Thuật toán tiếp tục di chuyển đến trạng thái kề cận tốt nhất và lặp lại quá trình này cho đến khi không tìm thấy trạng thái kề cận nào cải thiện được hoặc đạt đến một tiêu chí dừng.

Một số thuật toán tìm kiếm cục bộ phổ biến bao gồm:

* + Hill Climbing: Thuật toán Hill Climbing di chuyển lên các trạng thái có giá trị hàm mục tiêu tốt hơn. Nó duyệt qua các trạng thái kề cận và chọn trạng thái tốt nhất để di chuyển đến. Tuy nhiên, Hill Climbing có thể bị mắc kẹt ở các giải pháp cục bộ tốt nhất và không thể tìm ra giải pháp toàn cục.
  + Simulated Annealing: Simulated Annealing kết hợp việc khám phá các khu vực mới và tránh rơi vào các giải pháp cục bộ tốt nhất. Nó sử dụng một tham số gọi là "nhiệt độ" để điều chỉnh mức độ khám phá và khả năng chấp nhận các giải pháp kém hơn. Thuật toán cho phép chấp nhận các trạng thái kề cận xấu hơn với xác suất nhất định, giúp tránh rơi vào các giải pháp cục bộ.

## 4.3 Hill Climbing

Hill Climbing (leo đồi) là một thuật toán tìm kiếm cục bộ đơn giản trong lĩnh vực trí tuệ nhân tạo và tối ưu hóa. Nó tìm kiếm một giải pháp tốt hơn bằng cách di chuyển từ trạng thái hiện tại đến các trạng thái kề cận có giá trị hàm mục tiêu cao hơn.

Thuật toán Hill Climbing bắt đầu bằng việc tạo ra một giải pháp ban đầu hoặc tìm kiếm một giải pháp khả thi trong không gian trạng thái. Sau đó, nó sử dụng các quy tắc di chuyển để tạo ra các trạng thái kề cận và đánh giá giá trị của chúng bằng hàm mục tiêu. Thuật toán chọn trạng thái kề cận có giá trị hàm mục tiêu cao nhất và di chuyển đến đó.

Tuy nhiên, Hill Climbing có một số hạn chế. Nó có thể bị mắc kẹt ở các giải pháp cục bộ tốt nhất và không thể tìm ra giải pháp toàn cục. Điều này xảy ra khi không có trạng thái kề cận nào có giá trị tốt hơn trạng thái hiện tại. Thuật toán cũng không có khả năng quay lại các bước đã đi qua, do đó có thể bỏ qua các giải pháp tốt hơn nằm ở các "đồi" khác nhau.

Một số biến thể của thuật toán Hill Climbing như Steepest Ascent Hill Climbing và First-Choice Hill Climbing cố gắng giải quyết một số hạn chế của thuật toán gốc. Steepest Ascent Hill Climbing chọn trạng thái kề cận tốt nhất trong tất cả các trạng thái kề cận, trong khi First-Choice Hill Climbing chọn ngẫu nhiên một trạng thái kề cận tốt nhất nếu nó cải thiện so với trạng thái hiện tại.

Một cải tiến khác là Hill Climbing với khởi tạo ngẫu nhiên nhiều lần (Multiple Random Restarts) nơi thuật toán được chạy nhiều lần với các giải pháp khởi tạo ngẫu nhiên khác nhau, nhằm tăng khả năng tìm ra giải pháp tốt nhất.

Hill Climbing là một thuật toán đơn giản và dễ triển khai, nhưng nó thường được sử dụng trong các bài toán đơn giản và không quá phức tạp. Trong các bài toán phức tạp hơn, các thuật toán tìm kiếm cục bộ khác phức tạp hơn thường được sử dụng để đảm bảo khả năng tìm kiếm toàn cục tốt hơn.

## 4.4 Các vấn đề của thuật toán Hill Climbing

### 4.4.1 Quá nhiều successors

**Nguyên nhân:**

Vấn đề có quá nhiều successors của thuật toán Hill Climbing có thể gây ra bởi nhiều lý do:

* Không gian trạng thái lớn: Nếu không gian trạng thái của bài toán quá lớn, số lượng successor có thể tăng một cách đáng kể. Ví dụ, trong các bài toán tìm kiếm trạng thái hoặc bài toán tối ưu hóa với không gian lớn, số lượng trạng thái có thể làm cho số lượng successor trở nên quá lớn.
* Quy tắc di chuyển phức tạp: Trong một số bài toán, quy tắc di chuyển từ một trạng thái sang trạng thái khác có thể phức tạp và tốn nhiều thời gian tính toán. Điều này đặc biệt đúng khi các quy tắc di chuyển phụ thuộc vào các ràng buộc phức tạp hoặc phải tính toán lại toàn bộ hàm mục tiêu.

**Cách khắc phục**

Để khắc phục vấn đề trên, ta có thể áp dụng các phương pháp:

* Giới hạn số lượng successor: Thay vì duyệt qua tất cả các successor, bạn có thể áp dụng các kỹ thuật như Beam Search để chỉ duyệt qua một số lượng hạn chế các successor tốt nhất. Điều này giúp giảm tải tính toán và tối ưu hóa sử dụng tài nguyên.
* Sắp xếp và chọn lọc các successor: Áp dụng các phương pháp sắp xếp và lựa chọn để chỉ xem xét một số lượng hạn chế các successor quan trọng nhất hoặc tiềm năng nhất. Điều này giúp tập trung vào các successor có tiềm năng cao và giảm bớt số lượng successor không quan trọng.
* Tối ưu hóa quy tắc di chuyển: Nếu quy tắc di chuyển giữa các trạng thái phức tạp và tốn nhiều thời gian tính toán, bạn có thể cân nhắc tối ưu hóa quy tắc di chuyển. Điều này có thể bao gồm việc sử dụng thuật toán tìm kiếm thông minh để giảm thời gian tính toán của quy tắc hoặc áp dụng các phương pháp tối ưu hóa khác.

Tùy thuộc vào bài toán cụ thể, một hoặc nhiều phương pháp trên có thể được kết hợp để giảm vấn đề quá nhiều successor và tối ưu hóa hiệu suất của thuật toán Hill Climbing.

### 4.4.2 Local optimum

Local optimum (cực tiểu/cực đại cục bộ) là trạng thái tốt nhất trong các trạng thái lân cận mà thuật toán Hill Climbing có thể đạt được. Khi thuật toán Hill Climbing bị mắc kẹt tại một local optimum, nó không thể tiến xa hơn trong việc tìm kiếm giải pháp tối ưu toàn cục.

***Nguyên nhân***

Nguyên nhân chính của việc Hill Climbing mắc kẹt tại local optimum là do thiếu khả năng khám phá rộng rãi và bị hạn chế trong việc di chuyển ra khỏi các vùng trạng thái không tối ưu. Thuật toán chỉ di chuyển đến trạng thái kề cận tốt nhất mà nó tìm thấy, và nếu không có trạng thái kề cận nào cải thiện được giải pháp hiện tại, nó sẽ kết thúc và cho rằng đó là giải pháp tối ưu.

***Cách khắc phục***

* + Random Restart Hill Climbing: Thay vì bắt đầu từ một điểm khởi tạo duy nhất, thuật toán được thực hiện nhiều lần với các điểm khởi tạo ngẫu nhiên khác nhau. Điều này giúp thuật toán thoát khỏi các local optimum bằng cách khởi động lại và tìm kiếm từ các điểm khởi tạo khác nhau.
  + Steepest Ascent Hill Climbing: Thay vì chỉ di chuyển đến trạng thái kề cận đầu tiên mà nó gặp, thuật toán tìm kiếm tất cả các trạng thái kề cận và chọn trạng thái tốt nhất. Điều này giúp tăng khả năng khám phá và tránh rơi vào các local optimum.
  + Simulated Annealing: Sử dụng một cơ chế ngẫu nhiên để chấp nhận các trạng thái kề cận không tốt nhằm tránh rơi vào local optimum. Thuật toán duy trì một nhiệt độ và điều chỉnh mức độ chấp nhận của các trạng thái không tốt theo thời gian.
  + Genetic Algorithms: Sử dụng khái niệm về di truyền và đột biến để khám phá không gian trạng thái rộng hơn. Các giải pháp được biểu diễn dưới dạng cá thể và tiến hóa thông qua quá trình chọn lọc, lai ghép và đột biến.

Bằng cách kết hợp các phương pháp này hoặc sử dụng các biến thể của thuật toán Hill Climbing, có thể tăng khả năng tránh local optimum và tìm kiếm giải pháp tốt hơn trong không gian tìm kiếm.

## 4.5 Thuật toán Simulated Annealing

Simulated Annealing là một thuật toán tìm kiếm metaheuristic được lấy cảm hứng từ quá trình làm nguội từ nhiệt độ cao đến nhiệt độ thấp trong quá trình rèn kim loại. Thuật toán này được sử dụng để tìm kiếm giải pháp xấp xỉ cho các bài toán tối ưu hóa có không gian trạng thái lớn và có thể có nhiều local optimum.

***Cách thức hoạt động của thuật toán:***

* + Khởi tạo trạng thái ban đầu: Bắt đầu với một trạng thái ban đầu, có thể được chọn ngẫu nhiên hoặc thông qua một phương pháp khởi tạo khác.
  + Thiết lập tham số: Đặt các tham số cho thuật toán, bao gồm nhiệt độ ban đầu, tốc độ làm nguội (cooling rate) và số lần lặp tối đa.
  + Vòng lặp: Bắt đầu vòng lặp, thuật toán sẽ thay đổi trạng thái hiện tại bằng cách chọn một trạng thái kề cận ngẫu nhiên.
  + Đánh giá hàm mục tiêu: Tính toán giá trị của hàm mục tiêu cho trạng thái mới.
  + Quyết định chuyển tiếp: Nếu trạng thái mới tốt hơn (đối với bài toán tối ưu hóa), nó sẽ trở thành trạng thái hiện tại. Ngược lại, nếu trạng thái mới không tốt hơn, có một xác suất nhất định rằng nó sẽ được chấp nhận dựa trên nhiệt độ hiện tại.
  + Làm nguội: Giảm nhiệt độ theo tốc độ làm nguội đã được đặt trước. Quá trình làm nguội giảm xác suất chấp nhận các trạng thái mới không tốt, điều này giúp thuật toán di chuyển từ việc khám phá rộng sang khám phá sâu hơn.
  + Kiểm tra điều kiện dừng: Thuật toán dừng khi đạt đến điều kiện dừng được định trước, chẳng hạn như đạt đến số lần lặp tối đa hoặc nhiệt độ đã giảm đủ.
  + Trả về kết quả: Trả về trạng thái tốt nhất đã tìm thấy trong quá trình tìm kiếm.

Simulated Annealing cho phép thuật toán di tránh khỏi local optimum bằng cách chấp nhận một số trạng thái mới không tốt dựa trên xác suất và nhiệt độ. Quá trình làm nguội giúp giảm dần khả năng chấp nhận các trạng thái mới không tốt và tập trung vào khám phá các vùng tốt hơn trong không gian tìm kiếm.

## Searching in none-deterministic environments

### 4.6.1 Tìm kiếm trong môi trường không xác định

Thuật toán tìm kiếm trong môi trường không xác định là một phương pháp để tìm kiếm giải pháp trong một môi trường mà chúng ta không biết trước và có sự không chắc chắn. Ý tưởng chính của thuật toán này là sử dụng tìm kiếm thông qua không gian trạng thái để khám phá và đưa ra quyết định dựa trên thông tin thu thập được.

* + Khởi tạo: Bước đầu tiên của thuật toán là khởi tạo trạng thái ban đầu của môi trường và mục tiêu cần đạt được. Trạng thái ban đầu có thể chứa thông tin về vị trí, trạng thái của các đối tượng, và bất kỳ thông tin nào khác cần thiết.
  + Tạo cây tìm kiếm: Thuật toán tạo một cây tìm kiếm để tìm kiếm qua các trạng thái khác nhau của môi trường. Mỗi nút trong cây biểu diễn một trạng thái của môi trường và các hành động có thể được thực hiện từ trạng thái đó.
  + Mở rộng cây tìm kiếm: Thuật toán tiếp tục mở rộng cây tìm kiếm bằng cách thực hiện các hành động có thể từ trạng thái hiện tại. Các hành động này có thể làm thay đổi trạng thái của môi trường và dẫn đến các trạng thái mới.
  + Kiểm tra mục tiêu: Khi mở rộng cây tìm kiếm, thuật toán kiểm tra xem trạng thái hiện tại có đạt được mục tiêu hay không. Nếu trạng thái đạt được mục tiêu, thuật toán kết thúc và trả về chuỗi hành động dẫn đến mục tiêu. Ngược lại, thuật toán tiếp tục mở rộng cây tìm kiếm.
  + Xác định chiến lược: Trong quá trình mở rộng cây tìm kiếm, thuật toán phải xác định chiến lược để chọn hành động tiếp theo từ một trạng thái. Có nhiều chiến lược khác nhau có thể được sử dụng, bao gồm tìm kiếm theo chiều sâu, tìm kiếm theo chiều rộng, tìm kiếm thông minh (như thuật toán A\*), và nhiều hơn nữa. Chiến lược được chọn phụ thuộc vào bản chất của môi trường và yêu cầu của vấn đề cụ thể.
  + Quyết định và khám phá: Trong quá trình tìm kiếm, thuật toán phải đưa ra quyết định dựa trên thông tin thu thập được từ các trạng thái đã khám phá. Thông qua việc khám phá và mở rộng cây tìm kiếm, thuật toán có thể tìm ra giải pháp tốt nhất hoặc ít nhất là một giải pháp chấp nhận được dựa trên mục tiêu và các ràng buộc của môi trường.

Thuật toán tìm kiếm trong môi trường không xác định có thể có nhiều biến thể và phụ thuộc vào bản chất cụ thể của vấn đề. Tuy nhiên, ý tưởng cơ bản là sử dụng tìm kiếm thông qua không gian trạng thái để khám phá và đưa ra quyết định dựa trên thông tin thu thập được từ các trạng thái đã khám phá.

### 4.6.2 Thuật toán And-Or Graph Search

Thuật toán And-Or Graph Search (tìm kiếm đồ thị and-or) là một phương pháp tìm kiếm trong môi trường không xác định hoặc môi trường có sự phụ thuộc giữa các tình huống. Nó được sử dụng để tìm kiếm các chuỗi hành động có thể đạt được mục tiêu trong một môi trường có sự tương tác giữa các tình huống.

And-or graph search là một phương pháp mở rộng của thuật toán tìm kiếm trong không gian trạng thái. Nó sử dụng đồ thị để biểu diễn các trạng thái và các quan hệ giữa chúng. Đồ thị này bao gồm các nút và các cạnh, trong đó mỗi nút đại diện cho một trạng thái và mỗi cạnh đại diện cho một hành động.

Thuật toán and-or graph search hoạt động bằng cách mở rộng cây tìm kiếm và kiểm tra các ràng buộc and-or trong quá trình tìm kiếm. Dưới đây là mô tả chi tiết về ý tưởng của thuật toán and-or graph search:

* + Khởi tạo: Bước đầu tiên của thuật toán là khởi tạo trạng thái ban đầu của môi trường và mục tiêu cần đạt được. Trạng thái ban đầu có thể chứa thông tin về vị trí, trạng thái của các đối tượng và bất kỳ thông tin nào khác cần thiết.
  + Tạo đồ thị: Thuật toán tạo một đồ thị với các nút đại diện cho các trạng thái của môi trường và các cạnh đại diện cho các hành động có thể thực hiện từ các trạng thái đó. Các cạnh có thể được gắn kết với ràng buộc and hoặc or, phụ thuộc vào các quan hệ giữa các trạng thái.
  + Mở rộng đồ thị: Thuật toán tiếp tục mở rộng đồ thị bằng cách thực hiện các hành động có thể từ trạng thái hiện tại. Các hành động này có thể làm thay đổi trạng thái của môi trường và dẫn đến các trạng thái mới.
  + Kiểm tra mục tiêu: Khi mở rộng đồ thị, thuật toán kiểm tra xem trạng thái hiện tại có đạt được mục tiêu hay không. Nếu trạng thái đạt được mục tiêu, thuật toán kết thúc và trả về chuỗi hành động dẫn đến mục tiêu. Ngược lại, thuật toán tiếp tục mở rộng đồ thị.
  + Xác định chiến lược: Trong quá trình mở rộng đồ thị, thuật toán phải xác định chiến lược để chọn hành động tiếp theo từ một trạng thái. Có nhiều chiến lược khác nhau có thể được sử dụng, bao gồm tìm kiếm theo chiều sâu, tìm kiếm theo chiều rộng, tìm kiếm thông minh (như thuật toán A\*) và nhiều hơn nữa. Chiến lược được chọn phụ thuộc vào bản chất của môi trường và yêu cầu của vấn đề cụ thém.
  + Quyết định và khám phá: Trong quá trình tìm kiếm, thuật toán and-or graph search đưa ra quyết định dựa trên thông tin thu thập được từ các trạng thái đã khám phá. Nó khám phá các lựa chọn hành động và cập nhật đồ thị để tiếp tục tìm kiếm. Thuật toán sẽ tìm kiếm qua các nút và cạnh của đồ thị để xác định các chuỗi hành động dẫn đến mục tiêu.

Thuật toán and-or graph search cho phép mô hình hóa các ràng buộc phức tạp trong môi trường không xác định. Nó xử lý các quan hệ and-or giữa các trạng thái và hành động, cho phép tìm kiếm các chuỗi hành động phù hợp với các ràng buộc này.

Tuy nhiên, việc thực hiện thuật toán and-or graph search có thể phức tạp và đòi hỏi các phép toán tính toán lớn. Việc xác định các quan hệ and-or và xử lý các ràng buộc trong môi trường không xác định có thể là một thách thức. Đồng thời, việc thiết kế và triển khai thuật toán phụ thuộc vào bản chất cụ thể của vấn đề và môi trường được xem xét.

Trên thực tế, có nhiều biến thể và cải tiến của thuật toán and-or graph search được phát triển để xử lý các vấn đề cụ thể và cải thiện hiệu suất tìm kiếm. Các biến thể này thường tối ưu hóa các phép toán và quy trình tìm kiếm để đạt được kết quả tốt hơn trong môi trường không xác định.

## 4.7 Searching in partially observable environments

### 4.7.1 No observation

Tìm kiếm mà không có quan sát, còn được gọi là tìm kiếm mù hoặc tìm kiếm không có cảm biến, đề cập đến quá trình tìm kiếm trong trí tuệ nhân tạo mà trong đó một agent khám phá không gian vấn đề mà không có bất kỳ thông tin hoặc quan sát về trạng thái hoặc môi trường hiện tại. Đây là một nhiệm vụ đầy thách thức vì agent không có phản hồi hoặc gợi ý để hướng dẫn hành động.

Các thuật toán tìm kiếm mù dựa vào sự kết nối và cấu trúc của không gian vấn đề để khám phá và tìm kiếm một giải pháp. Những thuật toán này tạo ra và đánh giá theo cách hệ thống các trạng thái hoặc đường đi khả thi cho đến khi tìm thấy trạng thái mục tiêu hoặc giải pháp. Vì không có thông tin về trạng thái hiện tại, các thuật toán tìm kiếm mù xem xét tất cả các hành động có thể và khám phá các đường đi khác nhau để đạt được mục tiêu.

Có nhiều thuật toán tìm kiếm mù phổ biến mà có thể được sử dụng tùy thuộc vào đặc điểm và ràng buộc của vấn đề. Một số thuật toán tìm kiếm mù phổ biến bao gồm:

* + Tìm kiếm rộng trước (Breadth-First Search - BFS): BFS khám phá tất cả các trạng thái con của trạng thái hiện tại trước khi đi sâu vào các trạng thái con tiếp theo. Nó thường được sử dụng để tìm kiếm giải pháp ngắn nhất.
  + Tìm kiếm sâu trước (Depth-First Search - DFS): DFS duyệt qua các trạng thái con của trạng thái hiện tại cho đến khi không còn trạng thái nào khả thi, sau đó quay lui và thử các lựa chọn khác. DFS thường được sử dụng trong các không gian vấn đề lớn và không có giới hạn về bộ nhớ.
  + Tìm kiếm theo chi phí đồng nhất (Uniform Cost Search): Uniform Cost Search duyệt qua các trạng thái con với chi phí thấp nhất trước. Nó thích hợp cho các bài toán có chi phí khác nhau cho mỗi hành động.
  + Tìm kiếm tốt nhất (Best-First Search): Best-First Search sử dụng một hàm đánh giá để ước lượng sự tốt đẹp của mỗi trạng thái và lựa chọn trạng thái có giá trị đánh giá tốt nhất. Điều này cho phép thuật toán tìm kiếm tiến tới giải pháp tiềm năng nhanh hơn.
  + Tìm kiếm theo chi phí tham lam (Greedy Search): Greedy Search lựa chọn trạng thái dựa trên hàm đánh giá mục tiêu mà không xem xét các hành động tiếp theo. Nó tập trung vào việc tiến tới mục tiêu mà không quan tâm đến việc có tìm ra giải pháp tối ưu hay không.

Các thuật toán tìm kiếm mù phù hợp khi không có thông tin có sẵn về miền vấn đề hoặc khi không gian vấn đề nhỏ đến mức có thể được khám phá hoàn toàn. Tuy nhiên, chúng có thể trở nên không hiệu quả hoặc không khả thi đối với không gian vấn đề lớn hơn hoặc phức tạp hơn. Trong những trường hợp như vậy, việc kết hợp heuristics hoặc kiến thức cụ thể về miền vấn đề có thể giúp hướng dẫn quá trình tìm kiếm và cải thiện hiệu suất, dẫn đến các thuật toán tìm kiếm thông thái hoặc thuật toán tìm kiếm heuristics.

### 4.7.2 Partially observable

Tìm kiếm trong môi trường một phần quan sát là một vấn đề thách thức trong trí tuệ nhân tạo. Trong môi trường này, tác nhân không có thông tin đầy đủ về trạng thái hiện tại và chỉ nhận được thông tin hạn chế hoặc không chắc chắn về trạng thái.

Để giải quyết vấn đề này, có một số kỹ thuật thường được sử dụng trong AI:

* + Tìm kiếm trạng thái tin cậy: Trong quyết định Markov một phần quan sát (POMDP), tác nhân sử dụng một trạng thái tin cậy để biểu diễn sự không chắc chắn về trạng thái. Các thuật toán tìm kiếm trạng thái tin cậy, như thuật toán lập kế hoạch Monte Carlo một phần quan sát (POMCP), sử dụng các phương pháp mô phỏng để ước lượng trạng thái tin cậy và thực hiện tìm kiếm trong không gian trạng thái tin cậy.
  + Lọc hạt: Lọc hạt, hay còn được gọi là phương pháp Monte Carlo tuần tự, là một kỹ thuật được sử dụng để ước lượng trạng thái tin cậy trong môi trường một phần quan sát. Trạng thái tin cậy được biểu diễn dưới dạng một tập hợp các hạt có trọng số, trong đó mỗi hạt đại diện cho một giả thuyết về trạng thái. Các thuật toán lọc hạt, ví dụ như bộ lọc Monte Carlo, cập nhật liên tục trạng thái tin cậy bằng cách sử dụng các quan sát và thực hiện tìm kiếm dựa trên ước lượng trạng thái tin cậy.
  + Mô hình Markov ẩn (HMMs): HMMs là mô hình xác suất thường được sử dụng để biểu diễn môi trường một phần quan sát. HMMs bao gồm các trạng thái ẩn, các ký hiệu quan sát, và các xác suất chuyển đổi và xác suất phát xạ. Tìm kiếm trong HMMs liên quan đến ước lượng chuỗi trạng thái ẩn có khả năng cao nhất dựa trên các quan sát, thường sử dụng các thuật toán như thuật toán Viterbi hoặc thuật toán thuận-lùi.
  + Thu thập thông tin: Trong môi trường một phần quan sát, tác nhân có thể cần tích cực thu thập thông tin để giảm độ không chắc chắn và cải thiện quá trình ra quyết định. Các kỹ thuật như khám phá tích cực, trong đó tác nhân chọn hành động để thu thập thông tin hữu ích, hoặc quan sát tích cực, nơi tác nhân tác động lên môi trường để tiết lộ thông tin ẩn, có thể được sử dụng để hướng dẫn quá trình tìm kiếm.
  + Tìm kiếm dựa trên heuristics: Trong một số trường hợp, các heuristic hoặc kiến thức cụ thể về lĩnh vực có thể được sử dụng để hướng dẫn quá trình tìm kiếm trong môi trường một phần quan sát. Heuristics có thể giúp tác nhân đưa ra các ước lượng trạng thái và hành động tốt hơn dựa trên thông tin có sẵn.

Những kỹ thuật trên đều có ưu điểm và hạn chế riêng, và sự lựa chọn phụ thuộc vào bối cảnh cụ thể và mục tiêu của vấn đề. Việc tìm kiếm trong môi trường một phần quan sát vẫn là một lĩnh vực nghiên cứu tích cực trong trí tuệ nhân tạo, và có nhiều phương pháp và thuật toán khác nhau được phát triển để giải quyết vấn đề này.

## 4.8 Online search

### 4.8.1 So sánh offline search và online search

Khái niệm offline search và online search

* Offline search: Offline search đề cập đến các phương pháp tìm kiếm mà ở đó, thuật toán được chạy trên máy tính trước khi áp dụng vào môi trường thực. Mỗi hành động của agent có thể dự đoán trước kết quả dựa vào mô hình chuyển tiếp. Một số ví dụ điển hình của offline search bao gồm thuật toán Duyệt sâu (DFS), Duyệt rộng (BFS), và A\*.
* Online search: Ngược lại, online search là các phương pháp tìm kiếm mà ở đó, thuật toán được thực hiện đồng thời với việc học hỏi trong môi trường thực. Khi càng tiếp tục thực hiện, khả năng tìm kiếm của hệ thống sẽ càng trở nên thông minh hơn. Một ví dụ của online search có thể là việc dự báo thời tiết.

Bảng so sánh: (online search sẽ useful trong trường hợp: there is a penalty for, nondeterministic enviroments, no enviroment model)

|  |  |  |
| --- | --- | --- |
|  | Offline Search | Online Search |
| Đầu vào | -Node gồm 5 thành phần:   * Initial state * Possible actions: ACTIONS * Transition model: RESULT * Goal test: GOAL-TEST * Cost: STEP-COST, PATH-COST   Dữ liệu toàn diện về môi trường và các tác vụ. | -Node gồm 4 thành phần:   * Initial state * Possible actions: ACTIONS * Goal test: GOAL-TEST * Cost: STEP-COST, PATH-COST   - Dữ liệu từ môi trường thực tế trong quá trình hoạt động. |
| Đầu ra | Giải pháp hoàn chỉnh trước khi triển khai. | Học hỏi và thích nghi liên tục trong quá trình hoạt động. |
| Cách thức hoạt động | a | Học hỏi và thích nghi liên tục trong quá trình hoạt động. |
| Thuật toán ứng dụng | DFS, BFS, A\*, thuật toán tối ưu hóa dựa trên mô hình đã biết. | Các thuật toán học máy, học củng cố, thích ứng với dữ liệu mới. |
| Ứng dụng | Mô phỏng, tối ưu hóa dựa trên mô hình xác định, quy hoạch. | Dự báo thời tiết, điều khiển robot trong môi trường không xác định. |

### 4.8.2 Competitive ratio

Trong online search có 2 loại của path costs đó là:

* Actual Path Cost: Đây là chi phí mà người thực hiện (agent) phải chịu trong quá trình khám phá môi trường và tìm kiếm giải pháp. Nó bao gồm cả các bước thăm dò và xác định phương án giải quyết.
* Shortest path cost: Đây là chi phí tối thiểu cần thiết để tìm ra giải pháp, nghĩa là đường đi ngắn nhất có thể để đạt được mục tiêu.
* Competitive ratio = actual path cost / shortest path cost: là tỷ số giữa Chi phí Đường Đi Thực Tế so với Chi phí Đường Đi Ngắn Nhất. Đây là một tiêu chí đánh giá hiệu quả của thuật toán, cho thấy mức độ gần gũi của chi phí thực tế mà thuật toán sử dụng so với chi phí lý tưởng nhất.

### 4.8.3 Online depth first search agent

A close-up of a computer code

Description automatically generatedPseudocode

Mô tả thuật toán:

* Input: **Agent** nhận một cảm biến s' mà nó sử dụng để xác định trạng thái hiện tại.
* Các biến lưu trữ:
* **result**: Một bảng lưu trữ kết quả của các hành động từ các trạng thái.
* **untried**: Bảng liệt kê các hành động chưa được thử cho mỗi trạng thái.
* **unbacktracked**: Bảng liệt kê các trạng thái chưa được **backtrack** từ một trạng thái
* hiện tại.
* Kiểm tra mục tiêu: Nếu **s'** là trạng thái mục tiêu, thuật toán sẽ dừng.
* Trạng thái mới: Nếu **s'** là một trạng thái mới chưa có trong **untried**, thì danh sách
* các hành động có thể thực hiện tại **s'** sẽ được thêm vào.
* Lưu trạng thái và hành động: Nếu s không phải là null, nghĩa là đã có một trạng thái trước đó, thuật toán sẽ cập nhật result với trạng thái mới và hành động đã thực hiện để đến được **s'**.
* Thêm vào unbacktracked: Thêm s vào đầu của danh sách **unbacktracked** cho **s'**.
* Kiểm tra hành động chưa thử: Nếu không còn hành động nào chưa thử ở **s'**, thuật toán tiến hành **backtrack**.
* Backtracking:
* Nếu không còn trạng thái nào để **backtrack**, thuật toán sẽ dừng lại.
* Nếu còn, thuật toán sẽ chọn một hành động chưa thử từ **unbacktracked** để thực hiện.
* Chọn hành động kế tiếp: Nếu có hành động chưa thử, thuật toán sẽ chọn một hành động từ untried.
* Cập nhật trạng thái: Cập nhật trạng thái hiện tại sang s'.
* Kết quả: Trả về hành động a để thực hiện tiếp theo.

Completeness: thuật toán sẽ tìm được solution nếu bài toán đó có solution

Thời gian thực hiện của online search sẽ lâu hơn offline search.

### 4.8.4 Online A\* search

Ý tưởng cốt lõi của nó là đưa ra quyết định ở mỗi bước bằng cách chọn một trạng thái kế tiếp dựa trên chi phí ước tính, kết hợp giữa chi phí trực tiếp từ trạng thái hiện tại đến trạng thái kế tiếp (cost(s, s')) và ước tính hàm lượng giá từ trạng thái hiện tại đến mục tiêu còn lại (H(s')).

Ban đầu, H(s) được thiết lập là h(s), một ước tính hàm lượng giá của khoảng cách từ điểm khởi đầu. Hàm lượng giá này được cập nhật động khi tác nhân tiến triển, tích hợp thông tin thời gian thực và điều chỉnh đánh giá hàm lượng giá, từ đó làm tinh chỉnh quá trình tìm đường đi.

Phương pháp này cho phép tìm kiếm thông tin và hiệu quả hơn, hiệu quả cân bằng việc khám phá và khai thác trong môi trường chưa biết hoặc môi trường được biết một phần.

### 4.8.5 Learning real- Time A\* agent

A screenshot of a computer

Description automatically generatedPseudocode

Input: (s’) percept cho biết cái state hiện tại của enviroment.

Ouput: với mỗi state mà agent nghĩ mình đang ở đó, thì nó nên có action gì, hay nói cách khác là trả về một action đối với 1 state.

Mô tả thuật toán:

* Nhận dạng trạng thái hiện tại (s'): Thuật toán bắt đầu bằng việc xác định trạng thái hiện tại thông qua percept **s'**.
* Kiểm tra mục tiêu (GOAL-TEST): Nếu trạng thái hiện tại s' là trạng thái mục tiêu, thuật toán sẽ dừng lại bằng cách trả về stop.
* Xử lý trạng thái mới: Nếu **s'** là một trạng thái mới chưa từng được gặp (không có trong bảng **H**), thuật toán sẽ cập nhật bảng **H** với chi phí ước lượng mới, được lấy từ hàm heuristic **h(s').**
* Cập nhật kết quả và chi phí ước lượng:
  + Nếu s không phải là null, thuật toán sẽ cập nhật **result[s, a]** với trạng thái mới **s'**.
  + Cập nhật bảng **H[s]** với chi phí tối thiểu từ các hành động có thể thực hiện từ **s**, sử dụng hàm **LRTA\*-COST.**
* Chọn hành động: Chọn hành động **a** từ tập hợp các hành động có thể (**ACTIONS(s')**) sao cho hành động đó tối thiểu hóa chi phí được ước lượng bởi **LRTA\*-COST.**
* Thực hiện hành động và cập nhật trạng thái:
  + Gán **s** với giá trị của **s'** (để thể hiện việc chuyển trạng thái).
  + Trả về hành động a để thực hiện.
* Tính toán chi phí (**LRTA\*-COST**): Đây là hàm được gọi để tính toán chi phí từ trạng thái hiện tại s đến trạng thái mới **s'**, cộng với chi phí ước lượng từ trạng thái mới **s'**.

Completeness: thuật toán sẽ tìm được solution nếu bài toán đó có solution.

Thời gian thực hiện được cải thiện vì nó sử dụng estimated cost.

## 4.9 Constraint Satisfactoin Problems

### 4.9.1 Constraint satisfaction problems (CSP)

Trong CSP có 3 components:

* Tập Biến (Variables): Đây là tập hợp các yếu tố cần được gán giá trị trong quá trình giải quyết vấn đề.
* Miền Giá Trị (Domains): Mỗi biến trong tập biến có một miền giá trị tương ứng, định nghĩa phạm vi các giá trị mà biến đó có thể nhận.
* Ràng Buộc (Constraints): Đây là tập hợp các điều kiện hoặc quy tắc mà các gán giá trị cho các biến phải tuân theo để đạt được một giải pháp hợp lệ.

Solution của CSP: gán giá trị cho tất cả các biến với giá trị thuộc domains của nó sao cho thỏa mãn tất cả các ràng buộc.

### 4.9.2 Constraint graph

Nodes: là tất cả các biến.

Links: là 2 biến trong 1 ràng buộc bất kỳ

### 4.9.3 Constraint propagation

Bài toán ràng buộc (CSP) thường được giải quyết hiệu quả hơn các phương pháp tìm kiếm cổ điển nhờ khả năng thu hẹp miền giá trị của biến.

Ý tưởng chủ yếu là tận dụng các biến đã xác định chắc chắn một giá trị để hạn chế lựa chọn cho các biến liên quan. Quá trình này diễn ra thông qua việc áp dụng các ràng buộc để giảm số lượng giá trị khả dĩ trong miền giá trị của các biến khác, tạo ra hiệu ứng dây chuyền làm giảm không gian tìm kiếm. Khi mỗi biến đều chỉ còn một giá trị khả dĩ duy nhất trong miền của mình mà không vi phạm ràng buộc, một giải pháp cho CSP được xác định.

### 4.9.4 Local consistency (node, arc, path)

Trong các vấn đề về ràng buộc (CSPs), "local consistency" (tính nhất quán cục bộ) áp dụng cho một nhóm các biến nằm gần nhau. Khi nhóm này thỏa mãn tất cả các ràng buộc giữa chúng, chúng ta gọi là đạt được tính nhất quán cục bộ.

Cách hoạt động của nó là làm cho các biến này thỏa mãn ràng buộc, rồi lan truyền điều kiện thỏa mãn đó sang các biến xung quanh, qua đó loại bỏ những giá trị trong miền giá trị khiến cho biến không nhất quán.

### 4.9.5 AC- 3 Algorithm (Arc consistency )

A screenshot of a white sheet of paper with black text

Description automatically generatedPseudocode:

Input: một binary csp của một problem với các components (X, D, C)

Output: False nếu không thỏa mãn ràng buộc, True nếu thỏa mãn ràng buộc

Mô tả thuật toán:

Bước 1: Khởi tạo

**AC-3** là hàm chính nhận đầu vào là một **CSP** nhị phân với các thành phần **X, D, C** tương ứng với tập hợp các biến, tập hợp miền giá trị cho mỗi biến và tập hợp các ràng buộc.

Một hàng đợi queue được khởi tạo, chứa tất cả các cung (**arc**) trong **CSP**.

Bước 2: Xử lý hàng đợi

Trong khi queue không rỗng, thực hiện các bước sau:

Lấy ra cung đầu tiên từ hàng đợi, bao gồm hai biến **X\_i** và **X\_j.**

Bước 3: Kiểm tra và cập nhật miền giá trị

Gọi hàm **REVISE** với **X\_i** và **X\_j**.

Nếu **REVISE** trả về **true**, tức là miền giá trị của **X\_i** đã được cập nhật, kiểm tra miền giá trị **D\_i.**

Nếu **D\_i** rỗng, tức là không có giá trị nào thoả mãn ràng buộc, trả về **false** - báo hiệu có mâu thuẫn và **CSP** không thể giải được.

Bước 4: Cập nhật hàng đợi

Nếu **D\_i** không rỗng, thêm tất cả các cung liên quan đến **X\_i** và các biến hàng xóm (**trừ X\_j**) vào queue để kiểm tra tiếp.

Bước 5: Kết luận

Nếu hàng đợi trống mà không tìm thấy mâu thuẫn, tức là tất cả các cung đều thoả mãn, trả về **true**.

Hàm **REVISE**

**REVISE** là một hàm con được gọi trong **AC-3.**

Duyệt qua mỗi giá trị **x** trong **D\_i**, kiểm tra nếu không tồn tại giá trị **y** trong **D\_j** sao cho cặp giá trị (**x, y**) thoả mãn ràng buộc giữa **X\_i** và **X\_j.**

Nếu x không thoả mãn với bất kỳ **y** nào, **x** sẽ bị xóa khỏi **D\_i** và cờ **revised** được đặt thành **true**.

Hàm trả về giá trị của cờ **revised** để biết liệu miền giá trị có được cập nhật hay không.

### 4.9.6 Path consistency

Sử dụng cho các constraint 3 biến.

Chúng ta có một bộ 2 biến **{(Xi, Xj)}** được xem là path – consistency, khi ta có {**Xi = a, Xj = b**} sau đó ta tìm được giá trị cho Xm để thoản mãn constraints với {**Xi, Xm**} và {**Xm, Xj**}. Còn nếu không tìm được thì tồn tại cặp giá trị (**a, b**) làm cho nó inconsistent.

### 4.9.7 K-consistency (dạng tổng quát của local consistency)

Ý tưởng: lấy ra một tập có K -1 biến, sau đó tìm giá trị cho biến thứ K, nếu tìm được thì nó đạt K-consistency, nếu tìm không được thì nó không đạt K-consistency.

### 4.9.8 Global constraints

Một ràng buộc toàn cục là ràng buộc có thể áp dụng cho một số lượng lớn biến. Các ví dụ điển hình bao gồm Alldiff và Atmost.

* Alldiff đảm bảo rằng tất cả các biến có giá trị khác nhau, chẳng hạn như trong một hàng, cột hoặc ô vuông nhỏ của Sudoku, tất cả các số từ 1 đến 9 phải khác nhau.
* Atmost giới hạn số lượng lần xuất hiện tối đa của một giá trị cụ thể trong một tập hợp các biến.

### 4.9.9 Solving a Sodoku puzzle (AC – 3, Triplets)

Một cách tiếp cận là sử dụng Triplets trong khi áp dụng AC-3:

* Bước 1: Xác định trong một đơn vị Sudoku, ví dụ một hàng, cột hoặc ô vuông 3x3, ba ô mà miền giá trị của chúng là tập con của một tập hợp X bao gồm ba giá trị.
* Bước 2: Sau đó, loại bỏ ba giá trị này khỏi miền giá trị của các ô còn lại trong cùng một đơn vị để thu hẹp các khả năng và dần đạt đến lời giải.

### 4.9.10 Backtracking search algorithm

Bản chất: DFS. Cải tiến hơn DFS, do backtracking sử dụng thuộc tính giao hoán

A screenshot of a computer program

Description automatically generatedPseudocode:

Input: csp

Output: solution hoặc failure

Mô tả thuật toán:

Khởi tạo Tìm Kiếm: **BACKTRACKING-SEARCH(csp**) là hàm khởi tạo thuật toán với csp là bài toán cần giải. Hàm này trả về kết quả của hàm **BACKTRACK** khi được gọi với một biến gán rỗng và csp.

Hàm lui tổng quát: **BACKTRACK(assignment, csp)** là hàm đệ quy thực hiện thuật toán lui. Nó nhận phân công hiện tại và csp, trả về một giải pháp hoặc thất bại.

Kiểm tra biến gán hoàn tất: Nếu biến gán hiện tại là hoàn tất (tức là tất cả các biến đều đã được phân giá trị), hàm trả về phân công đó như là một giải pháp.

Chọn biến chưa phân giá trị: **SELECT-UNASSIGNED-VARIABLE(csp)** chọn một biến chưa được phân giá trị từ **csp**.

Duyệt qua các giá trị có thể:

**ORDER-DOMAIN-VALUES(var, assignment, csp)** sẽ trả về một thứ tự các giá trị để thử cho var.

Với mỗi giá trị có thể, kiểm tra xem giá trị đó có nhất quán với biến gán hiện tại không.

Thêm Giá Trị Vào biến gán: Nếu giá trị nhất quán, thêm {**var = value**} vào biến gán.

Suy Luận:

**INFERENCE(csp, var, value)** cố gắng suy luận thêm thông tin từ việc thêm {**var = value**}.

Nếu suy luận không thất bại, thêm các suy luận vào biến gán.

Đệ quy lui:

Gọi đệ quy **BACKTRACK** với biến gán mới và csp.

Nếu result không phải là thất bại, trả về result.

Trả về biến gán:

Nếu không tìm thấy giải pháp, loại bỏ {**var = value**} và các suy luận từ biến gán.

Nếu không có giá trị nào dẫn đến giải pháp, trả về thất bại.

### 4.9.11 Comments on backtracking search algorithm

Select next variable and value:

* Khởi tạo tìm kiếm Heuristic giá trị còn lại tối thiểu (Minimum Remaining Values): Ưu tiên chọn biến có ít giá trị khả dụng nhất trong miền của nó, tức là có ít lựa chọn nhất để gán.
* Độ Heuristic (Degree Heuristic): Chọn biến có nhiều ràng buộc với các biến khác nhất, tức là có nhiều liên kết nhất với các phần còn lại của mạng lưới ràng buộc.

Inference:

* + Heuristic giá trị ít hạn chế nhất (**Least-constraining-value**): Chọn giá trị cho biến sao cho nó để lại nhiều lựa chọn nhất cho các biến tiếp theo, nghĩa là giảm thiểu ảnh hưởng đến miền của các biến còn lại.

Backtracking strategies:

* + **Backjumping**: Thay vì lui trở lại một cách tuần tự, xác định tập hợp các gán giá trị có khả năng xung đột (**conflict-set**) và nhảy trở lại trực tiếp đến vị trí có thể đã tạo ra xung đột, bỏ qua các bước không liên quan.

### 4.9.12 Min-conflict algorithm.

A white paper with black text

Description automatically generatedPseudocode:

Input: csp, max\_step (có chức năng giới hạn số lần duyệt)

Ouput: solution hoặc failure

Mô tả thuật toán:

* Khởi Tạo:
  + **MIN-CONFLICTS(csp, max\_steps)** bắt đầu với hai tham số: csp, đại diện cho bài toán phân công ràng buộc cần giải, và **max\_steps**, đại diện cho số bước tối đa thuật toán sẽ thực hiện trước khi dừng lại và thông báo thất bại.
  + Một phân công hoàn chỉnh **current** được tạo ra như là điểm xuất phát cho csp.
* Vòng Lặp:
  + Thuật toán thực hiện vòng lặp từ 1 đến **max\_steps.**
  + Trong mỗi bước lặp, nó kiểm tra xem phân công hiện tại current có phải là giải pháp cho **csp** hay không. Nếu đúng, trả về **current** như là một giải pháp.
* Chọn Biến Xung Đột: Nếu **current** không phải là giải pháp, thuật toán chọn một biến var một cách ngẫu nhiên từ tập hợp các biến trong csp mà đang xung đột (nghĩa là không thoả mãn các ràng buộc của **csp**).
* Giảm Thiểu Xung Đột: Chọn một giá trị **v** cho var sao cho nó làm giảm thiểu số lượng xung đột. Nói cách khác, **v** là giá trị mà khi gán cho var sẽ tạo ra ít xung đột nhất với phần còn lại của phân công **current**.
* Cập Nhật Phân Công: Gán giá trị **v** được chọn vào var trong phân công **current**.
* Kết Thúc: Nếu vòng lặp đạt tới **max\_steps** mà không tìm thấy giải pháp, thuật toán thông báo thất bại.

### 4.9.13 Tree CSP solver

Đặc điểm: thuật toán này chỉ chạy được trên tree, nếu không phải là tree thì loại bỏ các node tạo thành vòng trong graph. Thời gian giải bài toán chỉ tính trên thời gian tuyến tính và không cần dùng backtrack.

A screenshot of a computer

Description automatically generatedPseudocode:

Input: csp

Output: solution hoặc failure

Mô tả thuật toán:

* + Khởi Tạo:
  + **TREE-CSP-SOLVER(csp)** được gọi với csp là bài toán phân công ràng buộc cần giải, bao gồm các thành phần **X, D, C**, lần lượt là biến, miền giá trị, và ràng buộc.
  + **n** là số lượng biến trong **X**.
  + **assignment** là một phân công rỗng.
* Chọn Gốc: **root** là bất kỳ biến nào trong X được chọn làm gốc của cây.
* Sắp Xếp Topological: X được sắp xếp topologically từ root bằng hàm **TOPOLOGICALSORT(X, root).**
* Nhất Quán Hóa Cung: Thực hiện vòng lặp từ **j = n** xuống đến 2, và áp dụng hàm **MAKE-ARC-CONSISTENT(PARENT(X\_j), X\_j)** cho mỗi cặp biến **X\_j** và cha mẹ của nó để đảm bảo tính nhất quán của cung. Nếu không thể làm cho cung nào đó nhất quán, hàm trả về failure.
* Gán Giá Trị: Sau đó, một vòng lặp khác từ **i** = 1 tới n được thực hiện để phân công giá trị cho mỗi biến **X\_i.** assignment**[X\_i]** sẽ được gán với bất kỳ giá trị nhất quán nào từ **D\_i.**
* Kiểm Tra Giải Pháp: Nếu không tìm thấy giá trị nhất quán cho bất kỳ biến nào, hàm cũng trả về failure.
* Trả Về Biến Gán: Nếu tất cả các biến đều được phân công mà không có xung đột, hàm trả về assignment trả về cuối cùng.

# 5. GAME “SOKOBANAI”

## 5.1 Mô tả trò chơi

Trò chơi được thiết kệ dựa trên trò Sokoban. Sokoban là một trò chơi giải đố cổ điển được phát triển tại Nhật Bản vào những năm 1980. Trò chơi được chơi trên một bảng hình vuông, trong đó mỗi hình vuông là một tầng. Người chơi điều khiển một nhân vật, có thể di chuyển theo các hướng lên, xuống, trái, phải. Trên bảng có các khối vuông, được đặt ở các vị trí khác nhau. Mục tiêu của trò chơi là đẩy tất cả các khối vuông đến các vị trí đích được đánh dấu trên bảng.

Cách chơi trò Sokoban khá đơn giản. Người chơi sử dụng các phím mũi tên trên bàn phím để di chuyển nhân vật. Khi nhân vật đứng cạnh một khối vuông, người chơi có thể nhấn phím mũi tên theo hướng mà họ muốn đẩy khối vuông. Khối vuông sẽ di chuyển theo hướng đó, miễn là không có chướng ngại vật

A screenshot of a video game

Description automatically generated

Ở đây chương trình của chúng ta được áp dụng các thuật toán AI đã học. Chúng ta có thể chọn các level khác nhau bằng cách sửa dụng các nút di chuyển qua trái qua phải để chon các level khác nhau. Chúng ta sử dụng phím cách để chọn các thuật toán khác nhau. Ở đây chương trình được áp dụng 2 thuật toán đó là Depth first search và A start. Muốn bắt đầu chơi ta chỉ cần bấm phím enter để bắt đầu. Bạn sẽ phải chờ 1 khoảng thời gian để thuật toán tìm ra được đường đi sao cho đẩy các hộp đến vị trí đích.

Đây là giao điện khi bắt đầu trò chơi

A screenshot of a game

Description automatically generated

Khi trò chơi bắt đầu robot sẽ tự động di chuyển theo đường đã được tìm ra bởi thuật toán mà ta đã chọn. Ở phía bên góc phải màn hình chúng ta sẽ có danh sách các bước đi cảu robot.

Đây là giao diện trò chơi sau khi robot đã đẩy các hộp vào đích .

A screenshot of a video game

Description automatically generated

Chúng ta bấm phím **Enter** để để quay lại màn hình chính ban đầu để chọn lại level hoặc thuật toán khác.

## 5.2 Mô tả các class và function

### 5.2.1 Class state

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 get\_line(self):

if *self*.state\_parent is None:

return [*self*.board]

return (*self*.state\_parent).get\_line() + [*self*.board]

def compute\_heuristic(self):

list\_boxes = find\_boxes\_position(*self*.board)

if *self*.heuristic == 0:

*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

#### 5.2.1.1 Phương thức khởi tạo (\_\_init\_\_):

Phương thức này khởi tạo một đối tượng state với bốn tham số: **self**, **board**, **state\_parent**, và **list\_check\_point**.

**self.board** được gán bằng tham số board, có thể đại diện cho trạng thái hiện tại của bảng trò chơi.

**self.state\_parent** được thiết lập để state\_parent, có lẽ ám chỉ trạng thái trước đó của trò chơi, ngụ ý một liên kết đến một nước đi trước hoặc cấu hình trong một chuỗi các nước đi.

**self.cost** được khởi tạo với giá trị 1, có thể đại diện cho chi phí di chuyển từ state\_parent đến trạng thái hiện tại.

**self.heuristic** được đặt thành 0, khởi tạo giá trị ước lượng, được sử dụng trong các thuật toán tìm kiếm để ước tính chi phí để đạt được mục tiêu từ trạng thái hiện tại.

**self.check\_points** được tạo ra bằng cách thực hiện một bản sao chép sâu của list\_check\_point, đảm bảo rằng thay đổi self.check\_points không ảnh hưởng đến danh sách gốc được truyền vào. Danh sách này để đại diện cho các vị trí cụ thể hoặc điều kiện cần kiểm tra hoặc đáp ứng trong bảng trò chơi.

#### 5.2.1.2 Phương thức Get Line (get\_line):

Phương thức này nó trả về kết quả của phương thức get\_line() của self.board ghép với [self.board].

#### 5.2.1.3 Phương thức Tính Heuristic (compute\_heuristic):

Phương thức này bắt đầu bằng việc gọi find\_boxes\_position(self.board) để lấy vị trí của các hộp trên bảng, gán kết quả cho list\_boxes. Điều này ngụ ý rằng trò chơi liên quan đến việc di chuyển các hộp đến các vị trí nhất định, tương tự như trò chơi Sokoban. Sau đó, nó kiểm tra xem self.heuristic có bằng không không và, nếu có, nó tính toán giá trị heuristic. Giá trị heuristic được tính toán là tổng của giá trị tuyệt đối của sự chênh lệch giữa các tọa độ x của hộp và các điểm kiểm tra tọa độ x cộng với sự chênh lệch giữa các tọa độ y và các điểm kiểm tra tọa độ y cho tất cả các hộp.

### 5.2.2 Depth first search

def DFS\_search(board, list\_check\_point):

start\_time = time.time()

if spf.check\_win(board, list\_check\_point):

print("Found win")

return [board]

start\_state = spf.state(board, None, list\_check\_point)

list\_state = [start\_state]

list\_visit = [start\_state]

while len(list\_visit) != 0:

now\_state = list\_visit.pop()

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)

if spf.is\_board\_exist(new\_board, list\_state):

continue

if spf.is\_board\_can\_not\_win(new\_board, list\_check\_point):

continue

if spf.is\_all\_boxes\_stuck(new\_board, list\_check\_point):

continue

new\_state = spf.state(new\_board, now\_state, list\_check\_point)

if spf.check\_win(new\_board, list\_check\_point):

print("Found win")

return (new\_state.get\_line(), len(list\_state))

list\_state.append(new\_state)

list\_visit.append(new\_state)

end\_time = time.time()

if end\_time - start\_time > spf.TIME\_OUT:

return []

print("Not Found")

return []

Khởi tạo:

* Hàm bắt đầu với việc ghi lại thời gian bắt đầu để theo dõi thời gian thực hiện thuật toán.
* Kiểm tra xem trạng thái ban đầu đã đạt được mục tiêu hay chưa bằng cách sử dụng phương thức check\_win. Nếu đã thắng, hàm sẽ in ra thông báo và trả về bảng trò chơi hiện tại.
* Tạo ra một trạng thái ban đầu start\_state sử dụng bảng trò chơi hiện tại và danh sách các điểm kiểm tra.
* Khởi tạo hai danh sách: list\_state chứa danh sách các trạng thái đã thăm và list\_visit chứa danh sách các trạng thái sẽ được duyệt.

Vòng Lặp DFS: khi danh sách list\_visit không rỗng, vòng lặp sẽ:

* + Lấy ra một trạng thái (now\_state) từ đỉnh của ngăn xếp (danh sách list\_visit) để duyệt.
  + Tìm vị trí hiện tại của người chơi trên bảng trò chơi.
  + Lấy danh sách các vị trí mà người chơi có thể di chuyển đến từ vị trí hiện tại.
  + Duyệt qua các vị trí có thể di chuyển: Với mỗi vị trí có thể di chuyển đến, hàm thực hiện:
    - Tạo một bảng trò chơi mới dựa trên nước đi tiếp theo.
    - Kiểm tra xem bảng mới này đã được duyệt trước đó hay chưa.
    - Kiểm tra xem bảng mới có tiềm năng đạt chiến thắng hay không.
    - Kiểm tra xem tất cả các hộp có bị kẹt không, nếu có thì bỏ qua các trạng thái này.
    - Tạo một trạng thái mới dựa trên bảng mới và trạng thái hiện tại.
    - Kiểm tra xem trạng thái mới có phải là trạng thái thắng cuộc không. Nếu có, in ra thông báo và trả về kết quả.
    - Nếu không: Thêm trạng thái mới vào danh sách đã thăm (list\_state) và danh sách sẽ được duyệt tiếp (list\_visit).

Kiểm soát thời gian: Sau mỗi lần lặp, hàm kiểm tra thời gian đã trôi qua. Nếu thời gian thực hiện vượt quá giới hạn (được định nghĩa trong spf.TIME\_OUT), hàm sẽ dừng và trả về một danh sách rỗng, cho thấy việc tìm kiếm đã vượt quá thời gian cho phép mà không tìm thấy giải pháp.

### 5.2.3 A\* search

def AStar\_Search(board, list\_check\_point):

start\_time = time.time()

if spf.check\_win(board, list\_check\_point):

print("Found win")

return [board]

start\_state = spf.state(board, None, list\_check\_point)

list\_state = [start\_state]

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 = spf.move(now\_state.board, next\_pos,

cur\_pos, list\_check\_point)

if spf.is\_board\_exist(new\_board, list\_state):

continue

if spf.is\_board\_can\_not\_win(new\_board, list\_check\_point):

continue

if spf.is\_all\_boxes\_stuck(new\_board, list\_check\_point):

continue

new\_state = spf.state(new\_board, now\_state, list\_check\_point)

if spf.check\_win(new\_board, list\_check\_point):

print("Found win")

return (new\_state.get\_line(), len(list\_state))

list\_state.append(new\_state)

heuristic\_queue.put(new\_state)

end\_time = time.time()

if end\_time - start\_time > spf.TIME\_OUT:

return []

end\_time = time.time()

if end\_time - start\_time > spf.TIME\_OUT:

return []

print("Not Found")

return []

Khởi tạo:

* Hàm bắt đầu với việc ghi lại thời gian bắt đầu để theo dõi thời gian thực hiện thuật toán.
* Kiểm tra xem trạng thái ban đầu đã đạt được mục tiêu hay chưa bằng cách sử dụng phương thức check\_win. Nếu đã thắng, hàm sẽ in ra thông báo và trả về bảng trò chơi hiện tại.
* Tạo ra một trạng thái ban đầu start\_state sử dụng bảng trò chơi hiện tại và danh sách các điểm kiểm tra.
* Khởi tạo danh sách: list\_state chứa danh sách các trạng thái đã thăm.
* Sử dụng hàng đợi ưu tiên (PriorityQueue) cho việc lựa chọn trạng thái tiếp theo. Hàng đợi ưu tiên sẽ lưu trữ các trạng thái có độ ưu tiên cao nhất. Độ ưu tiên của một trạng thái được tính bằng tổng của chi phí di chuyển từ trạng thái ban đầu đến trạng thái đó và giá trị heuristic của trạng thái đó. Giá trị heuristic của một trạng thái là một ước lượng về độ gần của trạng thái đó đến trạng thái chiến thắng.

Vòng lặp chính : Bước này lặp cho đến khi hàng đợi rỗng.

* Lấy ra một trạng thái (now\_state) có độ ưu tiên cao nhất ra khỏi hàng đợi.
* Tìm vị trí hiện tại của người chơi trên bảng trò chơi.
* Lấy danh sách các vị trí mà người chơi có thể di chuyển đến từ vị trí hiện tại.
* Duyệt qua các Vị Trí Có Thể Di Chuyển: Với mỗi vị trí có thể di chuyển đến, hàm thực hiện:
* Tạo một bảng trò chơi mới dựa trên nước đi tiếp theo.
* Kiểm tra xem để loại bỏ các bảng không mong muốn.
* Tạo một trạng thái mới dựa trên bảng mới và trạng thái hiện tại.
* Kiểm tra xem trạng thái mới có phải là trạng thái thắng cuộc không. Nếu có, in ra thông báo và trả về kết quả.
* Sau đó thêm trạng thái mới vào danh sách đã thăm (list\_state) và danh sách hàng đợi ưu tiên

Cuối cùng kiểm tra thời gian thực thi. Nếu thời gian thực thi vượt quá thời gian giới hạn, hàm sẽ trả về một mảng rỗng.

### 5.2.4 Các hàm hỗ trợ

#### 5.2.4.1 Kiểm tra các trạng thái chiến thắng

def check\_win(board, list\_check\_point):

for p in list\_check\_point:

if board[p[0]][p[1]] != 'b':

return False

return True

Input : Nhận vào 1 ma trận và danh sách điểm đích

Output: Trả về true nếu các điểm hộp trong bảng trùng với các điểm đích

Chức năng : Kiểm tra xem ma trận này đã thắng hay chưa

#### 5.2.4.2 Tìm vị trí người chơi

def find\_position\_player(board):

for x in range(len(board)):

for y in range(len(board[0])):

if board[x][y] == 'p':

return (x, y)

return (-1, -1)

Input: Ma trận kiểm tra

Output: Trả về vị tri của người chơi

Chức năng: Duyệt qua các vị trí trong ma trận nếu vị trí đó là ‘p’ trả về vị trí đó

#### 5.2.4.3 Tìm vị trí các hộp

def find\_boxes\_position(board):

result = []

for i in range(len(board)):

for j in range(len(board[0])):

if board[i][j] == 'b':

result.append((i, j))

return result

Input: Ma trận kiểm tra

Output: Trả về danh sách vị tri của các hộp

Chức năng: Duyệt qua các vị trí trong ma trận nếu vị trí đó là ‘b’ thì thêm vào danh sach vị trí các hộp.

#### 5.2.4.4 Tìm các vị trí người chơi có thể di chuyển

def get\_next\_pos(board, cur\_pos):

'''return list of positions that player can move to from current position'''

x, y = cur\_pos[0], cur\_pos[1]

list\_can\_move = []

*# MOVE UP (x - 1, y)*

if 0 <= x - 1 < len(board):

value = board[x - 1][y]

if value == ' ' or value == 't':

list\_can\_move.append((x - 1, y))

elif value == 'b' and 0 <= x - 2 < len(board):

next\_pos\_box = board[x - 2][y]

if next\_pos\_box != '1' and next\_pos\_box != 'b':

list\_can\_move.append((x - 1, y))

*# MOVE DOWN (x + 1, y)*

if 0 <= x + 1 < len(board):

value = board[x + 1][y]

if value == ' ' or value == 't':

list\_can\_move.append((x + 1, y))

elif value == 'b' and 0 <= x + 2 < len(board):

next\_pos\_box = board[x + 2][y]

if next\_pos\_box != '1' and next\_pos\_box != 'b':

list\_can\_move.append((x + 1, y))

*# MOVE LEFT (x, y - 1)*

if 0 <= y - 1 < len(board[0]):

value = board[x][y - 1]

if value == ' ' or value == 't':

list\_can\_move.append((x, y - 1))

elif value == 'b' and 0 <= y - 2 < len(board[0]):

next\_pos\_box = board[x][y - 2]

if next\_pos\_box != '1' and next\_pos\_box != 'b':

list\_can\_move.append((x, y - 1))

*# MOVE RIGHT (x, y + 1)*

if 0 <= y + 1 < len(board[0]):

value = board[x][y + 1]

if value == ' ' or value == 't':

list\_can\_move.append((x, y + 1))

elif value == 'b' and 0 <= y + 2 < len(board[0]):

next\_pos\_box = board[x][y + 2]

if next\_pos\_box != '1' and next\_pos\_box != 'b':

list\_can\_move.append((x, y + 1))

return list\_can\_move

Input : Ma trận và vị trí của người chơi

Output: Danh sach các vị trí có thể di chuyển

Chức năng:Hàm này sẽ kiểm tra những vị trí nao mà ngừoi chơi có thể di chuyển

#### 5.2.4.5 Hàm di chuyển

def move(board, next\_pos, cur\_pos, list\_check\_point):

new\_board = assign\_matrix(board)

if new\_board[next\_pos[0]][next\_pos[1]] == 'b':

x = 2\*next\_pos[0] - cur\_pos[0]

y = 2\*next\_pos[1] - cur\_pos[1]

new\_board[x][y] = 'b'

new\_board[next\_pos[0]][next\_pos[1]] = 'p'

new\_board[cur\_pos[0]][cur\_pos[1]] = ' '

for p in list\_check\_point:

if new\_board[p[0]][p[1]] == ' ':

new\_board[p[0]][p[1]] = 't'

return new\_board

Input: Ma trận ,vị trí tiếp theo của người chơi ,vị trí hiện tại của người chơi và danh sach sách điểm kiểm tra

Output :Ma trận mới sau khi áp dụng di chuyển

Chức năng : Hàm **move** này chịu trách nhiệm cập nhật bảng trò chơi theo nước đi được chọn, di chuyển người chơi và các hộp nếu cần, và đánh dấu các điểm kiểm tra.

#### 5.2.4.6 Hàm main

Hàm main trong đoạn mã này là điểm khởi đầu khi chạy chương trình game Sokoban.

* Khởi tạo Pygame và Cấu hình: Hàm main bắt đầu bằng việc khởi tạo Pygame và thiết lập cấu hình ban đầu như tạo cửa sổ hiển thị, khởi tạo font chữ, và cài đặt đồng hồ để theo dõi thời gian.
* Tải Dữ liệu: Các bảng trò chơi và điểm kiểm tra được tải từ các tệp tin định dạng .txt, và các tài nguyên đồ họa như hình ảnh cho người chơi, tường, hộp, điểm đích, và không gian trống được tải vào.
* Thiết lập Trò Chơi: Hàm sokoban được gọi để chạy trò chơi. Trong đó, trạng thái của trò chơi được quản lý thông qua biến sceneState với các trạng thái khác nhau như "init" cho việc chọn bản đồ, "loading" cho việc tải trò chơi, "executing" cho việc chạy thuật toán tìm đường, và "playing" khi đang hiển thị trò chơi.
* Xử lý Sự Kiện: Trong vòng lặp chính của trò chơi, sự kiện từ bàn phím được xử lý để điều khiển chuyển đổi giữa các bản đồ, chọn thuật toán giải quyết, và bắt đầu hoặc kết thúc trò chơi.
* Hiển thị Đồ Họa: Các hàm như initGame, loadingGame, foundGame, và notfoundGame được sử dụng để hiển thị các màn hình khác nhau trong quá trình chơi game, từ màn hình chính đến thông báo khi tìm thấy hoặc không tìm thấy giải pháp.
* Chạy Trò Chơi: Khi trò chơi bắt đầu, màn hình sẽ hiển thị các bước di chuyển của người chơi theo thuật toán đã chọn, và cuối cùng thông báo kết quả khi hoàn thành hoặc khi không tìm được lối đi trong thời gian cho phép.

# TÀI LIỆU THAM KHẢO

* Trần Nhật Quang (2023). *Bài giảng môn Trí tuệ nhân tạo.*
* *Understanding Search Algorithm in Artificial Intelligence.* November 30, 2023. <https://www.section.io/engineering-education/understanding-search-algorithms-in-ai/>
* *Time/Space Complexity of Depth First Search - Stack Overflow*. November 30, 2023. <https://stackoverflow.com/questions/36479640/time-space-complexity-of-depth-first-search>
* *Đạo văn – Wikipedia tiếng Việt*. November 30, 2023. <https://vi.wikipedia.org/wiki/%C4%90%E1%BA%A1o_v%C4%83n>
* *Một bản lịch sử đầy đủ về AI - Trí tuệ nhân tạo. (2023)*. November 30, 2023, từ [trituenhantao.io/kien-thuc/mot-ban-lich-su-day-du-ve-ai/](https://trituenhantao.io/kien-thuc/mot-ban-lich-su-day-du-ve-ai/)
* *Trí tuệ nhân tạo là gì? Cơ hội việc làm khi học trí tuệ nhân tạo .... (2023).* November 30, 2023, từ <https://iec.ut.edu.vn/vi/tri-tue-nhan-tao-la-gi-co-hoi-viec-lam-khi-hoc-tri-tue-nhan-tao-ra-sao/>