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

**KHOA CÔNG NGHỆ THÔNG TIN**

****

**ĐỒ ÁN HỌC PHẦN**

TÊN HỌC PHẦN: **Đồ Án Trí Tuệ Nhân Tạo**

MÃ SỐ LỚP HP: **[ARIN330585\_23\_1\_06CLC](https://fhqx.hcmute.edu.vn/course/view.php?id=50476" \o "Tri tue nhan tao_ Nhom 06CLC)**

Tên chủ đề : **ÁP DỤNG CÁC KIẾN THỨC ĐÃ HỌC ĐỂ XÂY DỰNG ỨNG DỤNG AI VÀ MÔ HÌNH GIẢI TRÍ**

**HỌC KỲ 1 NĂM 2023 - 2024**

**Giáo viên hướng dẫn: Trần Nhật Quang**

**Thực hiện: Nhóm 10**

|  |  |  |
| --- | --- | --- |
| **MSSV** | **Họ và tên** | **Mức độ đóng góp** |
| **21110298** | **Đặng Kim Thành** | **100%** |
| **21110279** | **Lê Minh Quang** | **100%** |
| **21110742** | **Võ Thị Minh Thục** | **100%** |
| **21119170** | **Ngô Nguyên Bảo** | **100%** |

**2**

**Thành Phố Hồ Chí Minh, tháng 12 năm 2023**

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

**KHOA CÔNG NGHỆ THÔNG TIN**

**ĐỀ TÀI ĐỒ ÁN HỌC PHẦN**

TÊN HỌC PHẦN: **Trí Tuệ Nhân Tạo**

MÃ SỐ LỚP HP: **[ARIN330585\_23\_1\_06CLC](https://fhqx.hcmute.edu.vn/course/view.php?id=50476" \o "Tri tue nhan tao_ Nhom 06CLC)**

Tên đề tài: **ÁP DỤNG CÁC KIẾN THỨC ĐÃ HỌC ĐỂ XÂY DỰNG ỨNG DỤNG AI VÀ MÔ HÌNH GIẢI TRÍ**

**Giảng viên giảng dạy: THẦY TRẦN NHẬT QUANG**

**Nhóm 10:**

**21110298 - Đặng Kim Thành**

**21110279 - Lê Minh Quang**

**21110742 - Võ Thị Minh Thục**

**21119170 - Ngô Nguyên Bảo**

**TÊN CÁC FILE SẢN PHẨM ĐỀ TÀI:**

**Tên thư mục đề tài: Nhom10\_AI\_Project.rar**

**Tập các thư viện sử dụng:**

# Speech: speech\_recognition, gtts, playsound, os, time, soundfile, sounddevice

# GUI: tkinter

# EDA: pandas, numpy, scipy, sklearn, ,matplotlib, seaborn, plotly.express

# GAME: pygame, random

#VIDEO-IMAGE: cv2, os

#IMAGE: PIL

**NHẬN XÉT CỦA GIẢNG VIÊN**

**.............................................................................................................................**

**.............................................................................................................................**

**.............................................................................................................................**

**.............................................................................................................................**

**.............................................................................................................................**

**.............................................................................................................................**

**.............................................................................................................................**

**.............................................................................................................................**

**.............................................................................................................................**

**Ngày……./……../202…..**

**Giảng viên: Ký tên**

**TP.HCM, ngày .... Tháng .... năm 2023**

**LỜI CẢM ƠN**

Em xin gửi lời cảm ơn đến Thầy Trần Nhật Quang, giảng viên dạy học phần Trí Tuệ Nhân Tạo tại trường Đại học Sư phạm Kỹ thuật Thành phố Hồ Chí Minh vì những kiến thức và sự hỗ trợ quý báu Thầy đã trao đến trong suốt quá trình học. Với sự tận tâm và tâm huyết của Thầy, em đã có cơ hội hoàn thiện và nắm vững kiến thức trong lĩnh vực này.

Đồ án học phần là một thử thách lớn đối với chúng em. Tuy nhiên, với sự chỉ dẫn và sự hỗ trợ từ Thầy, tuy có nhiều thiếu sót, em đã vượt qua mọi khó khăn và hoàn thành đồ án một cách thành công. Thầy đã luôn sẵn lòng giải đáp mọi thắc mắc của chúng em, cung cấp những lời khuyên quý giá và hướng dẫn chi tiết để chúng em có thể tiến bộ trong quá trình học tập.

Thầy không chỉ truyền đạt kiến thức một cách dễ hiểu mà còn khuyến khích chúng em khám phá, tư duy sáng tạo và ứng dụng lý thuyết vào thực tế. Thầy đã mang đến cho em những bài giảng thú vị và truyền cảm hứng cho em yêu thích lĩnh vực lập trình.

Cuối cùng, em muốn gửi lời cảm ơn chân thành đến Thầy Trần Nhật Quang vì những đóng góp to lớn của Thầy trong việc phát triển kiến thức và kỹ năng lập trình của em. Sự hỗ trợ và sự động viên từ Thầy đã giúp em vượt qua những thách thức và nâng cao khả năng của mình.

Em xin chân thành cảm ơn Thầy Trần Nhật Quang và mong rằng Thầy sẽ tiếp tục truyền đạt kiến cho nhiều thế hệ sinh viên khác nhau như em, mang lại sự ảnh hưởng tích cực và đóng góp cho sự phát triển của ngành lập trình.

Cuối cùng, em mong rằng Thầy luôn có thêm nhiều niềm vui và thành công trong công việc giảng dạy. Chúc Thầy luôn khỏe mạnh, hạnh phúc và tiếp tục truyền cảm hứng cho những người học tương lai.

Xin chân thành cảm ơn Thầy!

Trân trọng,

# DANH MỤC CÁC TỪ VIẾT TẮT

|  |  |
| --- | --- |
| CSDL hoặc DB | Cơ sở dữ liệu: DataBase |
| NSD = Users = Account | Người Sử Dụng = là quyền làm việc trên WebApp được đảm bảo bởi tối thiểu là 2 yếu tố: tên đăng nhập (UserName) và mật khẩu (Password) |
|  |  |
| AI | Artificial intelligence: Trí tuệ nhân tạo |
| EDA | Exploratory Data Analysis: Phân tích Khám phá Dữ liệu (Còn gọi là: phân tích dữ liệu thăm dò) |
| GUI | Graphical User Interface: Giao diện đồ hoạ |
| PYPRO | Lập trình Python |

**DANH MỤC CÁC THUẬT NGỮ**

|  |  |
| --- | --- |
| Giao diện người dùng:  User-Interface | Là hệ thống các màn hình giao tiếp cho phép người sử dụng tương tác với các thành phần phần mềm trong HTTT, thường là 1 trong các dong: Win-form, Web-form, Mobile-Form. |
| Tài khoản (Account) | Là một quyền làm việc trên HTTT được cấp phát cho một cá nhân thông qua tên tài khoản (username) và mật khẩu (password). |
| Trợ lý ảo: Voice Assistant | Là các hệ thống có khả năng “nghe” và “nói” với con người, nhờ đó hỗ trợ con người trong một số chức năng như một “trợ lý”. |

**MỤC LỤC**

[LỜI CẢM ƠN.........................................................................................................................4](#_Toc23752)

[DANH MỤC CÁC TỪ VIẾT TẮT 5](#_Toc11088)

[DANH MỤC CÁC THUẬT NGỮ 5](#_Toc14634)

[Phần 1: Tìm hiểu về đạo văn 8](#_Toc18747)

[1.1. Định nghĩa đạo văn 8](#_Toc14941)

[1.2. Những điều cần làm để tránh vấn đề đạo văn. 9](#_Toc22629)

[Phần 2: PHẦN LÝ THUYẾT 11](#_Toc27063)

[2.1.1. Python cơ bản 11](#_Toc29807)

[2.1.2. Python lists, function 12](#_Toc21870)

*[a. Lists (Danh sách)](#_Toc14186)* [12](#_Toc14186)

*[b. Functions (Hàm)](#_Toc10552)* [13](#_Toc10552)

*[c. Kết hợp Lists và Functions](#_Toc22648)* [13](#_Toc22648)

[2.2.4. Pakacges numpy, matplotlib, pandas 14](#_Toc6379)

[2.4.1. Local search – searching for goal states 20](#_Toc6246)

*[a. State Prediction (Dự Đoán Trạng Thái)](#_Toc5419)* [41](#_Toc5419)

*[b. Observation (Quan Sát)](#_Toc13359)* [41](#_Toc13359)

*[c. State Update (Cập Nhật Trạng Thái)](#_Toc16990)* [41](#_Toc16990)

[PHẦN 3: LẬP TRÌNH 53](#_Toc5258)

[3.1 LẬP TRÌNH GIAO DIỆN GAME 8 - PUZZLE. 53](#_Toc11402)

[3.1.1 Giới thiệu chung về chủ đề 53](#_Toc27375)

[3.1.1.1 Mô tả yêu cầu của chủ đề 53](#_Toc18113)

[3.1.1.2 Phân tích yều cầu của chủ đề 53](#_Toc12215)

[3.1.2 Các nền tảng kỹ thuật 53](#_Toc31557)

[3.1.2.1 Nền tảng kỹ thuật cơ bản của giao diện 53](#_Toc21055)

[3.1.2.2 Định nghĩa, hiệu suất của các thuật toán 54](#_Toc31123)

[3.1.2.3 Nền tảng kỹ thuật cơ bản của thuật toán 56](#_Toc13120)

[3.1.2.4 Hệ thống thư việc sử dụng 80](#_Toc6639)

[3.1.2.5 Thiết kế giao diện. 81](#_Toc689)

[3.1.2.6 Mã lệnh lập trình(python) - Full code 81](#_Toc10795)

[3.2 LẬP TRÌNH GIAO DIỆN GAME SOKOBAN 89](#_Toc9712)

[3.2.1 Giới thiệu chung về chủ đề 89](#_Toc21943)

[3.2.1.1 Mô tả yêu cầu của chủ đề 89](#_Toc12298)

[3.2.1.2 Phân tích yều cầu của chủ đề 89](#_Toc11083)

[3.2.2 Các nền tảng kỹ thuật 89](#_Toc5275)

[3.2.2.1 Nền tảng kỹ thuật cơ bản của giao diện 89](#_Toc19618)

[3.2.2.2 Nền tảng, kĩ thuật cơ bản của các hàm hỗ trợ 91](#_Toc30968)

[3.2.2.3 Định nghĩa, hiệu suất của các thuật toán 92](#_Toc9249)

[3.2.2.4 Nền tảng kỹ thuật cơ bản của thuật toán 93](#_Toc18904)

[3.2.2.5 So sánh biểu đồ ứng dụng của 2 thuật toán 94](#_Toc31758)

[3.2.2.6 Hệ thống thư việc sử dụng 97](#_Toc28809)

[3.2.2.7 Thiết kế giao diện. 97](#_Toc21040)

[Phần 4: KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN CỦA ĐỒ ÁN HỌC PHẦN 104](#_Toc3749)

[4.1 KẾT LUẬN 104](#_Toc6397)

[4.1.1 Những kết quả đạt được 104](#_Toc20223)

[4.1.2 Hạn chế. 104](#_Toc12619)

[4.2 HƯỚNG PHÁT TRIỂN 104](#_Toc13088)

[4.2.1 Hướng khắc phục các hạn chế 104](#_Toc21088)

[4.2.2 Hướng mở rộng ĐỒ ÁN HỌC PHẦN 104](#_Toc27028)

**Phần 1: Tìm hiểu về đạo văn**

* 1. ****Định nghĩa đạo văn****

Từ điển Merriam Webster định nghĩa hành vi đạo văn "Plagiarism" là; "Ăn cắp và chuyển giao ý tưởng hoặc lời nói của người khác như là của riêng mình". Nói một cách đơn giản, đạo văn là quá trình lấy lời nói hoặc ý tưởng của người khác và giả vờ rằng chúng là của riêng mình. Theo pháp luật Việt Nam tại điều 2 khoản “Đạo văn là việc sử dụng có hoặc không có chủ ý của tác giả các sản phẩm học thuật về các câu văn, đoạn văn, bài viết, số liệu, hình ảnh, thông tin và ý tưởng của người khác vào các sản phẩm của mình mà không có những chỉ dẫn/thừa nhận tác giả của những nội dung đã sử dụng.”

Đạo văn là một lỗi nghiêm trọng về tiêu chuẩn đạo đức khoa học, mà luật pháp (sở hữu trí tuệ) và các cơ sở đào tạo không thể bỏ qua. Đạo văn, tương tự như việc ăn cắp, không chỉ là hành động trái đạo đức mà còn là việc lấy mất sự cá nhân hóa và sáng tạo trong quá trình sáng tác. Khi chúng ta sao chép ý tưởng, công việc, và cấu trúc câu của người khác mà không đưa ra sự đóng góp cá nhân, chúng ta đang mất đi cơ hội để phát triển kỹ năng viết của bản thân. Nếu là sinh viên, trải nghiệm này có thể đánh mất sự hài lòng và tự hào khi đối diện với công trình mà mình đã ăn cắp. Đồng thời, việc đạo văn cũng phản ánh sự mù chữ về thông tin và thiếu tự tin trong khả năng diễn đạt suy nghĩ và ý tưởng cá nhân. Thay vì phát triển khả năng sáng tạo, đạo văn chỉ làm cho người viết trở nên phụ thuộc và thiếu độc lập trong quá trình học tập. Nó không chỉ là sự thừa nhận về khả năng mù chữ trong việc xử lý thông tin, mà còn là việc từ chối cơ hội để tỏa sáng và ghi dấu ấn cá nhân. Mặc dù có thể có những lợi ích ngắn hạn, như giảm áp lực hoặc tiết kiệm thời gian, nhưng hậu quả của đạo văn có thể rất đáng kể. Trong tương lai, khi sinh viên đối diện với thị trường lao động, khả năng sáng tạo và khả năng diễn đạt chính là những yếu tố quyết định sự thành công. Việc áp dụng kiến thức và kỹ năng một cách độc lập là chìa khóa để xây dựng sự nghiệp và đạt được thành công bền vững.

Theo pháp luật Việt Nam người được coi là đạo văn khi có các hành vi sau:

Sử dụng sản phẩm học thuật của người khác mà cam đoan rằng đó là của mình (được người viết thay tên); Sao chép (copy) quá nhiều từ một công trình (mặc dù có chỉ ra nguồn trích) để hình thành một phần lớn công trình của mình; Không dẫn nguồn đã trích khi thay đổi từ ngữ, di chuyển từ hoặc cụm từ, ý tưởng của tác giả khác; Cung cấp không chính xác về tác giả, nguồn của thông tin được trích dẫn, Sử dụng hơn 30% những sản phẩm học thuật của mình đã công bố vào những sản phẩm học thuật mới do mình là tác giả hoặc đồng tác giả mà không ghi rõ nguồn, gọi là tự đạo văn.

* 1. **Những điều cần làm để tránh vấn đề đạo văn.**

Hiểu rõ một số kiến thức cơ bản về vấn đề bản quyền.

Tham khảo nguồn tư liệu gốc.

Tìm hiểu kỹ về vấn đề mà bạn đang muốn nói tới.

Phân tích và tổng hợp thông tin.

Trích dẫn đoạn văn và nguồn của bài viết.

Diễn đạt lại nhiều lần bằng các cách khác nhau.

Kiểm tra lại bài viết của bạn.

Chúng em xin cam đoan dự án này do các thành viên trong 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.

Các thành viên trong nhóm:

Lê Minh Quang

Đặng Kim Thành

Võ Thị Minh Thục

Ngô Nguyên Bảo

**Phần 2: PHẦN LÝ THUYẾT**

**2.1. Tổng quan về Python**

Python, một ngôn ngữ lập trình nhập môn, được sáng tạo bởi Guido van Rossum với mục tiêu đơn giản và dễ hiểu. Cấu trúc của Python làm cho việc viết mã trở nên rõ ràng và giảm thiểu số lần gõ phím. Được đánh giá là lựa chọn lý tưởng cho người mới học lập trình, Python không chỉ mã nguồn mở mà còn hoàn toàn miễn phí. Sự tương thích đa nền tảng của Python là một ưu điểm lớn, với khả năng hoạt động trên Windows, macos, và Linux mà không gặp khó khăn. Python có khả năng nhúng và mở rộng, cho phép kết hợp với nhiều ngôn ngữ lập trình khác để phát triển ứng dụng phức tạp. Một điểm mạnh khác của Python là khả năng tự động chuyển đổi mã nguồn, giảm bớt lo lắng về quản lý bộ nhớ và dọn dẹp dữ liệu. Thư viện đồ sộ của Python đáp ứng mọi nhu cầu lập trình, mang lại sự thuận tiện và linh hoạt. Python không chỉ đơn giản hóa cho người mới học mà còn giữ nguyên bản chất hướng đối tượng. Với cách giải quyết vấn đề trực quan và dễ hiểu, Python trở thành một ngôn ngữ lập trình mạnh mẽ và linh hoạt cho cả người mới bắt đầu và những lập trình viên có kinh nghiệm.

**2.1.1. Python cơ bản**

A. Dòng Lệnh và thụt Lề:

Python sử dụng thụt lề để xác định khối mã. Điều này có nghĩa là bạn phải thụt lề các đoạn mã bên trong các khối như câu lệnh điều kiện, vòng lặp, hàm.

Thụt lề thường là 4 khoảng trắng hoặc một tab.

B. Biến và kiểu dữ liệu:

Không cần khai báo kiểu dữ liệu cho biến, Python tự động xác định kiểu dữ liệu.

Ví dụ:

X = 5 # Integer

Y = 3.14 # Float

Name = "John" # String

C. Nhận đầu vào từ người dùng:

Sử dụng hàm input() để nhận đầu vào từ người dùng.

Ví dụ:

User\_input = input("Nhập vào một giá trị: ")

D. Câu lệnh điều kiện:

Sử dụng if, elif, và else để kiểm tra điều kiện.

Ví dụ:

X = 10

If x > 0:

Print("x là số dương")

Elif x < 0:

Print("x là số âm")

Else:

Print("x là số không")

E. Vòng lặp:

Python hỗ trợ hai loại vòng lặp chính là for và while.

Ví dụ for:

For i in range(5):

Print(i)

Ví dụ while:

I = 0

While i < 5:

Print(i)

I += 1

G. Hàm:

Định nghĩa hàm bằng từ khóa def.

Ví dụ:

Def add(x, y):

Return x + y

H. List và Tuple:

List là một dạng dữ liệu có thể thay đổi, Tuple là một dạng dữ liệu không thể thay đổi.

Ví dụ:

My\_list = [1, 2, 3]

My\_tuple = (4, 5, 6)

K. Dictionary:

Dictionary là một cấu trúc dữ liệu chứa các cặp key-value.

Ví dụ:

My\_dict = {'name': 'John', 'age': 25}

L. Import Thư Viện:

Sử dụng import để đưa thư viện vào chương trình.

Ví dụ:

Import math

**2.1.2. Python lists, function**

Trong Python, danh sách (lists) là một trong những cấu trúc dữ liệu cơ bản và chức năng (function) là một khái niệm quan trọng.

1. *Lists (Danh sách)*

Danh sách trong Python là một chuỗi các giá trị có thể thay đổi được (mutable) và được sắp xếp theo thứ tự. Bạn có thể lưu trữ nhiều giá trị khác nhau trong một danh sách, và chúng có thể là các kiểu dữ liệu khác nhau.

# Tạo một danh sách

My\_list = [1, 2, 3, 'four', 5.0]

# Truy cập phần tử trong danh sách

Print(my\_list[0]) # Output: 1

# Thay đổi giá trị của một phần tử

My\_list[3] = 'four\_updated'

# Thêm phần tử vào cuối danh sách

My\_list.append(6)

# Xóa phần tử theo giá trị

My\_list.remove(2)

# Duyệt qua danh sách

For item in my\_list:

Print(item)

1. *Functions (Hàm)*

Hàm là một khối mã thực hiện một nhiệm vụ cụ thể và có thể được gọi từ nhiều nơi khác nhau trong chương trình. Trong Python, sử dụng từ khóa def để định nghĩa một hàm.

# Định nghĩa hàm

Def add\_numbers(a, b):

Return a + b

# Gọi hàm

Result = add\_numbers(3, 4)

Print(result) # Output: 7

1. *Kết hợp Lists và Functions*

# Hàm tính tổng của các số trong danh sách

Def calculate\_sum(numbers):

Total = sum(numbers)

Return total

# Sử dụng hàm với danh sách

My\_numbers = [1, 2, 3, 4, 5]

Result\_sum = calculate\_sum(my\_numbers)

Print(result\_sum) # Output: 15

Hàm Trả Về Nhiều Giá Trị và Gán Cho Nhiều Biến:

Max\_value, min\_value = get\_max\_min(my\_numbers)

Print("Max:", max\_value)

Print("Min:", min\_value)

**2.1.3. Python classes**

Giống như định nghĩa hàm bắt đầu bằng từ khóa def trong Python, định nghĩa lớp bắt đầu bằng từ khóa class. Class được bắt đầu bởi từ khoá class và theo sau từ khoá này là tên của class đó và dấu hai chấm. Những phần nằm dưới dòng chứa từ khoá class được gọi là class body. Nơi chúng ta sẽ định nghĩa class đó được khởi tạo như thế nào, có những phương thức và thuộc tính ra sao. Ví dụ chúng ta muốn khởi tạo class User ở hình thức đơn giản nhất.

Để định nghĩa một class User ở hình thức đơn giản nhất trong Python, bạn có thể sử dụng từ khóa class như sau:

Class User:

Def \_\_init\_\_(self, username, password):

Self.username = username

Self.password = password

# Tạo một đối tượng user

User1 = User(username="example\_user", password="password123")

# Truy cập thuộc tính của đối tượng

Print(user1.username) # In ra: example\_user

Print(user1.password) # In ra: password123

Trong ví dụ trên:

Class User: định nghĩa một class mới có tên là User.

Def \_\_init\_\_(self, username, password): là một phương thức khởi tạo (constructor) của class. Phương thức này được gọi khi một đối tượng mới của class được tạo. Self là tham số đại diện cho đối tượng được tạo, và username và password là các tham số được truyền vào khi tạo đối tượng.

Self.username và self.password là các thuộc tính của class, được sử dụng để lưu trữ thông tin về đối tượng. Trong trường hợp này, chúng ta lưu trữ thông tin về tên đăng nhập và mật khẩu của người dùng.

**2.2.4. Pakacges numpy, matplotlib, pandas**

*a. Numpy*

Mục đích: numpy cung cấp các đối tượng mảng nhiều chiều (ndarray) mạnh mẽ và các hàm để thực hiện các phép toán số học trên mảng đó. Nó làm cho việc làm việc với dữ liệu số lớn trở nên hiệu quả hơn.

Cài đặt: Bạn có thể cài đặt numpy bằng lệnh sau:

Pip install numpy

Sử dụng: Sau khi cài đặt numpy, bạn có thể tạo mảng, thực hiện phép toán số học, thống kê và nhiều thao tác khác trên dữ liệu số.

Import numpy as np

# Tạo một mảng numpy

Arr = np.array([1, 2, 3, 4, 5])

# Thực hiện các phép toán số học

Mean\_value = np.mean(arr)

*b. Matplotlib*

Mục đích: Matplotlib là một thư viện vẽ đồ thị mạnh mẽ, giúp bạn tạo ra các biểu đồ tĩnh, động và tương tác để hiển thị dữ liệu một cách trực quan.

Cài đặt: Bạn có thể cài đặt Matplotlib bằng lệnh sau:

Pip install matplotlib

Sử dụng: Bạn có thể sử dụng Matplotlib để vẽ đồ thị đơn giản hoặc phức tạp, histogram, scatter plots và nhiều loại biểu đồ khác.

Import matplotlib.pyplot as plt

# Vẽ đồ thị đơn giản

X = [1, 2, 3, 4, 5]

Y = [2, 4, 6, 8, 10]

Plt.plot(x, y)

Plt.show()

1. *Pandas*

Mục đích: Pandas cung cấp các cấu trúc dữ liệu linh hoạt như dataframe, Series, giúp bạn thực hiện các thao tác phức tạp trên dữ liệu, như lọc, nhóm, và ghép nối.

Cài đặt: Bạn có thể cài đặt Pandas bằng lệnh sau:

Pip install pandas

Sử dụng: Pandas thường được sử dụng để đọc dữ liệu từ các nguồn khác nhau (như CSV, Excel), xử lý và phân tích dữ liệu.

Import pandas as pd

# Tạo dataframe từ dict

Data = {'Name': ['Alice', 'Bob', 'Charlie'],

'Age': [25, 30, 35]}

Df = pd.dataframe(data)

# Hiển thị dataframe

Print(df)

Sau khi cài đặt các gói này, chúng ta có thể sử dụng chúng trong các đoạn mã Python hoặc notebook Jupyter của mình bằng cách nhập chúng:

Import numpy as np

Import matplotlib.pyplot as plt

Import pandas as pd

**2.2. Ethics and laws for AI**

**2.2.1. Ethics**

Theo diễn đàn kinh tế thế giới “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”

Trong triết học đạo đức (normative ethics), có ba phương pháp chính đề xuất các cơ sở hoặc tiêu chí để đánh giá đạo đức hành vi. Các phương pháp này đều tập trung vào việc xác định xem hành vi nào được xem là đạo đức và hành động nào được xem là không đạo đức. Các phương pháp này bao gồm:

Đạo đức luật (Deontological Ethics):

* Chủ nghĩa Deon (Deontology): Chủ nghĩa này xem xét tính đạo đức của hành vi dựa trên việc xem xét hành động đó theo các quy tắc hay nguyên tắc đạo đức.
* Immanuel Kant: Là một trong những nhà triết học nổi tiếng ủng hộ chủ nghĩa deon, Kant chủ trương rằng một hành động chỉ là đạo đức nếu nó tuân theo nguyên tắc đạo đức và có thể được áp dụng chung một cách công bằng cho mọi người.

Hậu đạo đức (Consequentialist Ethics):

* Chủ nghĩa Hậu quả (Consequentialism): Chủ nghĩa này đánh giá tính đạo đức của một hành động dựa trên kết quả của nó. Nếu hành động mang lại hậu quả tích cực, nó được xem là đạo đức.
* Utilitarianism: Là một dạng phổ biến của chủ nghĩa hậu quả, utilitarianism chấp nhận rằng hành động nên tối đa hóa hạnh phúc hoặc giảm thiểu khổ đau cho số lượng lớn người.

Đạo đức đo lường (Virtue Ethics):

* Chủ nghĩa Đức Tính (Virtue Ethics): Chủ nghĩa này tập trung vào đức tính của người thực hiện hành động hơn là vào hành động cụ thể đó. Nó đề xuất rằng để đạo đức, một người cần phát triển và thực hành các đức tính đạo đức.
* Aristotle: Là một trong những nhà triết học nổi tiếng ủng hộ chủ nghĩa đức tính, Aristotle chú trọng đến việc phát triển đức tính tốt thông qua thói quen và hành động đúng đắn.

**2.2.2. Data analytics and Law**

Vấn Đề (Issue): Việc sử dụng thuật toán máy học để đưa ra quyết định trong hệ thống tư pháp đang đặt ra nhiều thách thức về tính minh bạch, trách nhiệm và công bằng.

Quy Tắc (Rule): Áp dụng các quy định của pháp luật liên quan, ví dụ như quy định về quyết định tự động trong GDPR (Nghị định Bảo vệ Dữ liệu Chung châu Âu), cũng như các nguyên tắc đạo đức trong phát triển và triển khai trí tuệ nhân tạo.

Áp Dụng (Apply): Phân tích cách thuật toán máy học được sử dụng trong hệ thống tư pháp. Xem xét mức độ minh bạch của thuật toán, xử lý dữ liệu cá nhân, và cách quyết định được đưa ra. Đối chiếu với các yêu cầu về quyền riêng tư và công bằng trong quy định.

Kết Luận (Conclusion): Kết luận về việc liệu việc sử dụng trí tuệ nhân tạo trong hệ thống tư pháp tuân thủ các quy tắc pháp lý và nguyên tắc đạo đức. Đề xuất các biện pháp cụ thể để cải thiện tính minh bạch và công bằng, có thể bao gồm việc áp dụng các tiêu chuẩn chuẩn mực cho phát triển AI trong lĩnh vực pháp lý.

Luật pháp về Trí tuệ Nhân tạo (AI) có thể thay đổi tùy theo quốc gia và khu vực. Dưới đây là một số ví dụ về cách một số quốc gia đã đưa ra hoặc đang xem xét các quy định liên quan đến AI:

Châu Âu (Liên minh châu Âu): GDPR (Nghị định Bảo vệ Dữ liệu Chung châu Âu): Quy định về quyết định tự động và quyền riêng tư của cá nhân trong ngữ cảnh của AI.

Đề xuất về Luật Nhân Quyền Kỹ Thuật Số: Đang được xem xét, nhằm đảm bảo rằng quyền lợi của con người được bảo vệ trong môi trường AI.

Hoa Kỳ: Hiện tại, không có một đám mây luật pháp toàn diện về AI tại cấp quốc gia, nhưng một số tiểu bang và thành phố đã thực hiện các quy định đặc biệt, ví dụ như California với CCPA (California Consumer Privacy Act).

Trung Quốc: Quy định về Bảo Mật Dữ liệu và Quyền Riêng Tư: Đang ngày càng được mở rộng để bao gồm quy định về sử dụng AI và dữ liệu.

Canada: Luật Pháp về Quyền Riêng Tư: Canada có quy định về quyền riêng tư mạnh mẽ và tổ chức như Ủy ban Quyền Riêng Tư của Canada đã đưa ra hướng dẫn về việc áp dụng nguyên tắc quyết định tự động trong GDPR.

Ấn Độ: Dự Luật Dữ Liệu Cá Nhân (Dự Luật Quyền Lực Dữ Liệu): Nước này đang xem xét một số thay đổi lớn trong lĩnh vực bảo vệ dữ liệu, có thể bao gồm cả các quy định về trí tuệ nhân tạo.

Tính đến thời điểm kiến thức của mình đến tháng 1 năm 2022, Việt Nam chưa có một luật riêng biệt về trí tuệ nhân tạo (AI). Tuy nhiên, có một số văn bản pháp luật và quy định có thể liên quan đến việc sử dụng và phát triển trí tuệ nhân tạo:

Luật An Ninh Mạng (2015): Có các quy định liên quan đến bảo mật thông tin và an ninh mạng, một phần quan trọng trong bối cảnh sử dụng công nghệ AI.

Luật Bảo vệ Người Tiêu Dùng (2010): Có thể liên quan đến việc sử dụng dữ liệu cá nhân trong các ứng dụng AI, đặc biệt là những ứng dụng liên quan đến người tiêu dùng.

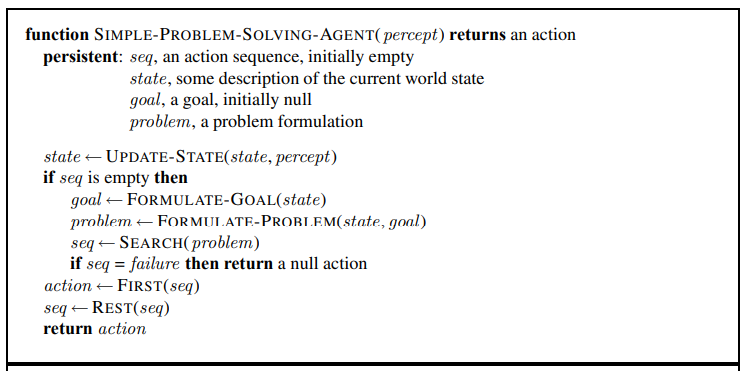
Luật Bảo Vệ Dữ Liệu Cá Nhân (2018): Quy định về bảo vệ dữ liệu cá nhân, có thể áp dụng cho việc thu thập và xử lý dữ liệu trong các hệ thống AI.

Chiến lược Quốc gia về Công Nghệ Thông Tin đến Năm 2020 và Chiến lược AI đến Năm 2030: Đây không phải là văn bản pháp luật, nhưng là các chiến lược quốc gia chỉ đạo cho phát triển và ứng dụng công nghệ thông tin, bao gồm cả trí tuệ nhân tạo.

**2.3. Fundamental ideas and concepts**

**2.3.1. Solving problems by searching**

Ý tưởng của giải thuậy: SIMPLE-PROBLEM-SOLVING-AGENT là tạo ra một tác nhân (agent) có khả năng giải quyết vấn đề trong một môi trường cụ thể, giải thuật này tập trung vào việc duy trì thông tin, cập nhật trạng thái, xác định mục tiêu, đặt vấn đề cụ thể, tìm kiếm giải pháp và thực hiện các hành động để đạt được mục tiêu. Điều này tạo ra một cách tiếp cận linh hoạt và tổ chức để giải quyết vấn đề trong môi trường có thể thay đổi.



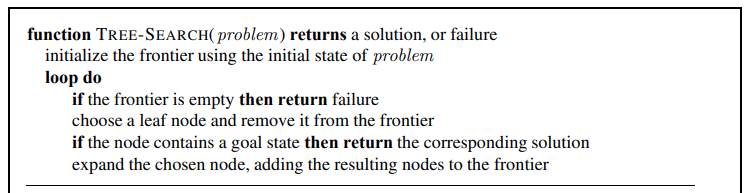
* persistent: seq, an action sequence, initially empty: Khởi tạo một biến seq để lưu trữ một chuỗi các hành động, và ban đầu nó là rỗng.
* state, some description of the current world state: Khởi tạo một biến state để mô tả trạng thái hiện tại của thế giới.
* goal, a goal, initially null: Khởi tạo một biến goal để đặt một mục tiêu, và ban đầu nó là null.
* problem, a problem formulation: Khởi tạo một biến problem để định nghĩa một vấn đề, một cách biểu diễn cụ thể của vấn đề cần giải quyết.
* state ← UPDATE-STATE(state, percept): Cập nhật trạng thái dựa trên thông tin môi trường mới nhận được (percept).
* if seq is empty then: Nếu chuỗi seq là trống.
* goal ← FORMULATE-GOAL(state): Đặt mục tiêu bằng cách sử dụng trạng thái hiện tại.
* problem ← FORMULATE-PROBLEM(state, goal): Xác định vấn đề cần giải quyết dựa trên trạng thái hiện tại và mục tiêu.
* seq ← SEARCH(problem): Tìm kiếm một chuỗi hành động để đạt được mục tiêu trong vấn đề đã định nghĩa.
* if seq = failure then return a null action: Nếu không tìm thấy chuỗi hành động, trả về một hành động null.
* action ← FIRST(seq): Lấy hành động đầu tiên từ chuỗi.
* seq ← REST(seq): Loại bỏ hành động đầu tiên từ chuỗi.
* return action: Trả về hành động đã chọn.

**2.3.2. Searching for solutions**

TREE-SEARCH:

Ý tưởng thuật toán: khởi tạo Frontier là initial state sau đó ta đi vào vòng lặp. Trong vòng lặp chọn 1 leaf node để mở rộng và loại nó ra khỏi frontier. Tiếp tục chọn các node đượcmở rộng là leaf node cho đến khi nào tìm được node là goal state hoặc Frontier rỗng mà không tìm được kết quả thì vòng lặp sẽ dừng.

Giải thuật:



Initialize the Frontier: sử dụng trạng thái ban đầu của vấn đề để khởi tạo biên (frontier).

Loop: bắt đầu một vòng lặp vô hạn.

Check if the Frontier is Empty: kiểm tra xem biên có trống không. Nếu có, trả về thất bại vì không có nút nào để kiểm tra.

Choose a Leaf Node and Remove It: chọn một nút lá từ biên và loại bỏ nó khỏi biên.

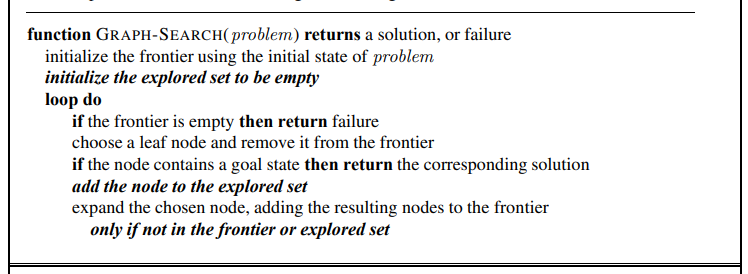
Check if the Node Contains a Goal State: kiểm tra xem nút được chọn có chứa trạng thái mục tiêu hay không. Nếu có, trả về giải pháp tương ứng.

Expand the Chosen Node: mở rộng nút đã chọn, thêm các nút kết quả vào biên.

GRAPH-SEARCH:

Ý tưởng thuật toán: Giống như Tree Search nhưng khi đã chọn 1 leaf node để mở rộng tathêm nó vô explored set (tập hợp các node đã được khám phá) để không phải chọn lại các node đó lần sau.

Giải thuật:



Initialize the Frontier: sử dụng trạng thái ban đầu của vấn đề để khởi tạo biên.

Initialize the Explored Set to Be Empty: khởi tạo tập đã khám phá để làm trống.

Loop: bắt đầu một vòng lặp vô hạn.

Check if the Frontier is Empty: kiểm tra xem biên có trống không. Nếu có, trả về thất bại vì không có nút nào để kiểm tra.

Choose a Leaf Node and Remove It: chọn một nút lá từ biên và loại bỏ nó khỏi biên.

Check if the Node Contains a Goal State: kiểm tra xem nút được chọn có chứa trạng thái mục tiêu hay không. Nếu có, trả về giải pháp tương ứng.

Add the Node to the Explored Set: thêm nút vào tập đã khám phá để đánh dấu rằng nó đã được xem xét.

Expand the Chosen Node: mở rộng nút đã chọn, thêm các nút kết quả vào biên nếu chúng không xuất hiện trong biên hoặc tập đã khám phá. Điều này giúp tránh sự lặp lại khi tìm kiếm.

**2.4. Searching in more complex environments**

**2.4.1. Local search – searching for goal states**

Local search là một phương pháp trong lĩnh vực trí tuệ nhân tạo, dùng để giải quyết vấn đề tìm kiếm cục bộ hay địa phương. Phương pháp này tập trung vào việc thăm dò trong một phạm vi hẹp của không gian trạng thái, chủ yếu tập trung vào trạng thái hiện tại hoặc những trạng thái lân cận.

Cách thức tìm kiếm:

Local search sử dụng chiến lược tìm kiếm tập trung vào một state hiện tại và những state gần đó.

Thay vì xây dựng toàn bộ không gian trạng thái, local search chỉ xem xét một phần nhỏ của nó, giúp giảm bớt áp lực về bộ nhớ.

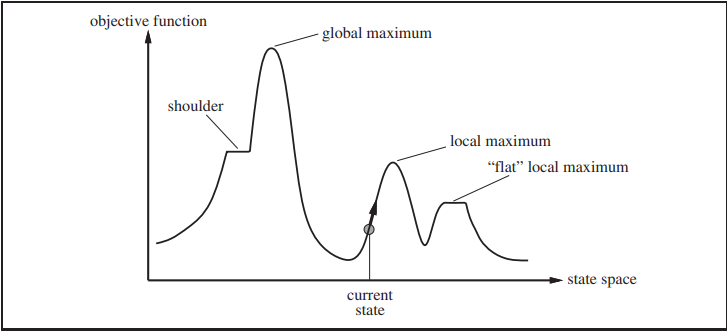
Mục tiêu của local search là tìm ra trạng thái mục tiêu (goal state) thỏa mãn điều kiện cụ thể của vấn đề.

Đặc điểm chính: Local search thường chỉ duy trì một current node (nút hiện tại) và xem xét các neighbor (trạng thái lân cận) của nó.

Do sử dụng ít bộ nhớ, local search phù hợp với giải quyết các vấn đề có không gian trạng thái lớn, thậm chí là vô hạn hoặc liên tục.

Local search đặc biệt hữu ích trong việc giải quyết các vấn đề liên quan đến không gian trạng thái vô hạn hoặc liên tục.

Với chiến lược tập trung, local search có thể tìm ra giải pháp tối ưu hoặc gần tối ưu trong một thời gian ngắn. Local search là một công cụ linh hoạt có thể áp dụng cho nhiều lĩnh vực, từ tối ưu hóa đường đi trong thành phố đến quy hoạch sản xuất trong môi trường công nghiệp.



**2.5. Uninformed search**

**2.5.1. Breadth-first search( BFS)**

**- Breadth-first search( BFS)** là thuật toán tìm kiếm theo chiều rộng, việc tìm kiếm gồm 2 thao tác:

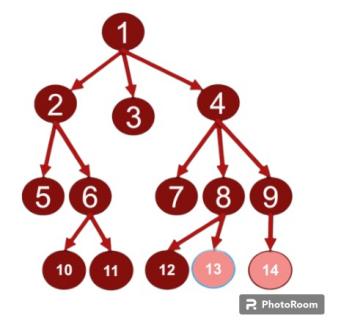
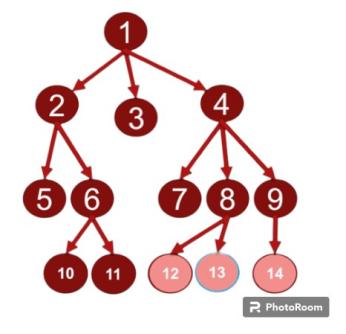
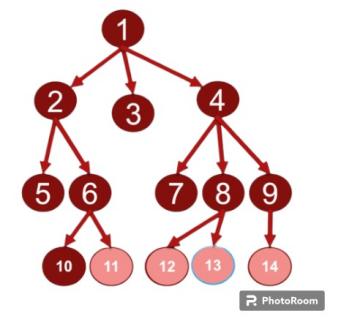
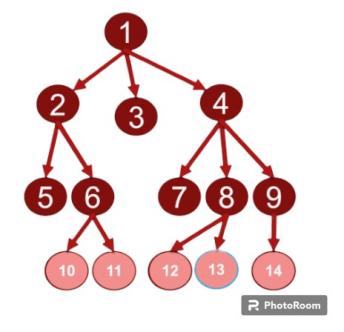
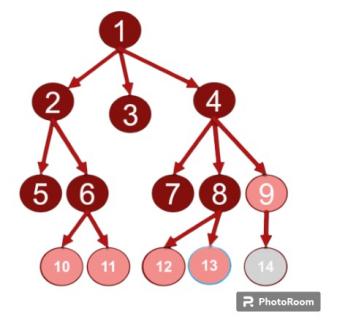
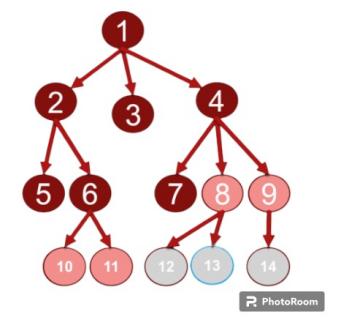
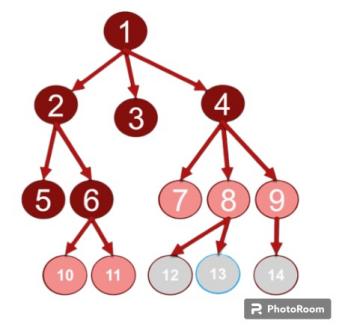
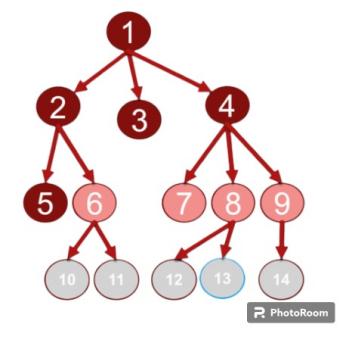
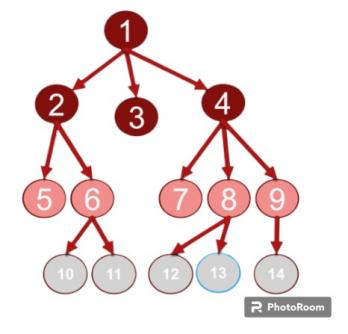
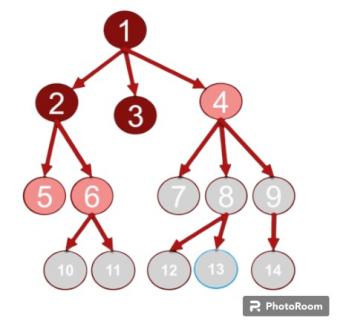
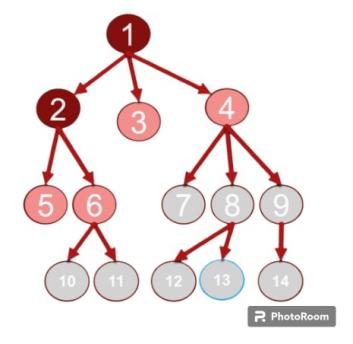
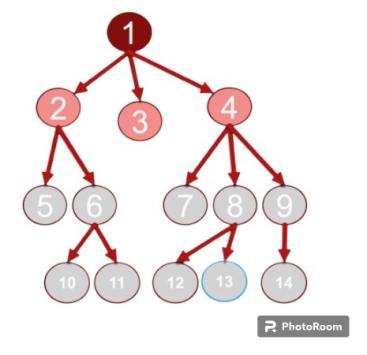
+ Thao tác 1: cho trước một đỉnh của đồ thị.

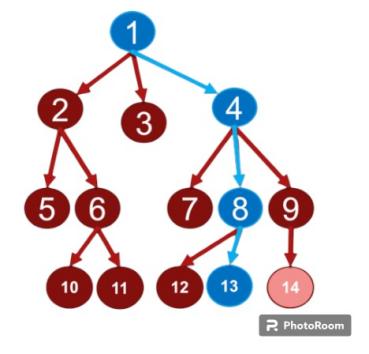
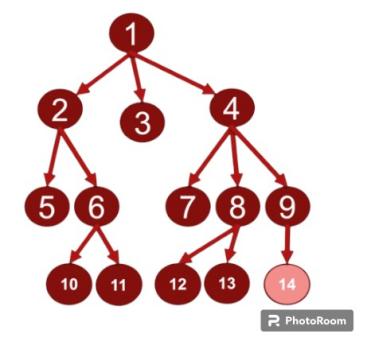
+ Thao tác 2: thêm các đỉnh liền kề với đỉnh đã cho vào danh sách mà có thể hướng tới tiếp theo.

- Mục đích sử dụng thuật toán **Breadth-first search( BFS):**

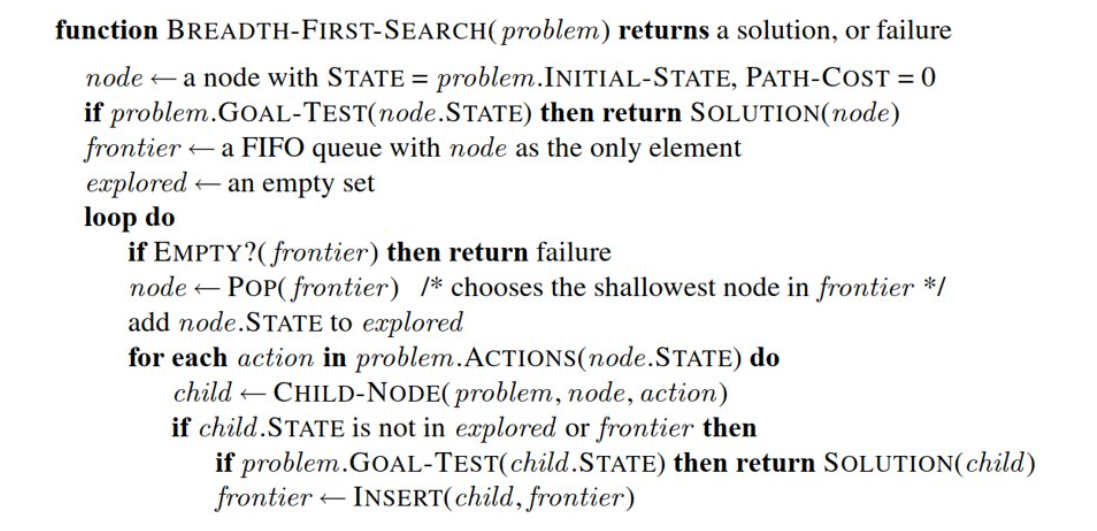
+ Tìm kiếm đường đi từ một đỉnh cho trước tới một đỉnh đích.

+ Tìm kiếm đường đi từ đỉnh gốc tới tất cả các đỉnh khác.





**Thuật toán Breadth-first search ( BFS):**



Các bài toán sửa dụng thuật toán **Breadth-first search( BFS)**

|  |
| --- |
| **def breadth\_first\_graph\_search(problem):**  **"""Bread first search (GRAPH SEARCH version)**  **See [Figure 3.11] for the algorithm"""**  **node = Node(problem.initial) # Tạo node là vị trí đầu**  **if problem.goal\_test(node.state): # Kiểm tra xem đã tới đích chưa**  **return node**  **frontier = deque([node]) # Bỏ node hiện tại vào queue**  **explored = set() # Thiết lập tệp lưu trữ rỗng**  **while frontier: # Nếu queue khác rỗng**  **node = frontier.popleft() # Bốc node ở đầu ra khỏi queue**  **explored.add(node.state) # Thêm vào tệp lưu trữ trạng thái**  **for child in node.expand(problem): # Bắt đầu xét các hậu duệ của vị trí hiện tại**  **if child.state not in explored and child not in frontier: # Xét xem hậu duệ đó đã từng được xét tới chưa**  **if problem.goal\_test(child.state): # Kiểm tra xem hậu duệ đó có phải là nơi cần đến không**  **return child**  **frontier.append(child) # Thêm hậu duệ vào cuối queue**  **return None # Nếu queue rỗng mà chưa tìm tới điểm đích thì trả về không tìm ra** |

- BFS có thể tìm ra giải pháp tối ưu nên chi phí đường đi là hàm không giảm theo độ sâu

**2.5.2. Uniform-cost search algorithm**

- Tìm kiếm chi phí thống nhất là một biến thể của thuật toán Dijikstra. Nó được sử dụng để tìm đường đi tối thiểu từ nút nguồn đến nút đích xung quanh biểu đồ có trọng số có hướng. Thuật toán tìm kiếm này sử dụng phương pháp tiếp cận mạnh mẽ, truy cập tất cả các nút dựa trên trọng số hiện tại của chúng và tìm đường dẫn có chi phí tối thiểu bằng cách kiểm tra liên tục tất cả các đường dẫn có thể.

- Vì nó không xem xét bất kỳ trạng thái hoặc thông tin trước đó nào về đường dẫn hoặc nút đích nên đây là một thuật toán tìm kiếm không chính xác.

- Sử dụng mảng boolean đã truy cập và hàng đợi ưu tiên để tìm chi phí tối thiểu. Nút có chi phí tối thiểu có mức độ ưu tiên cao nhất. Nó sử dụng tìm kiếm mù vì không có thông tin trước về các nút.

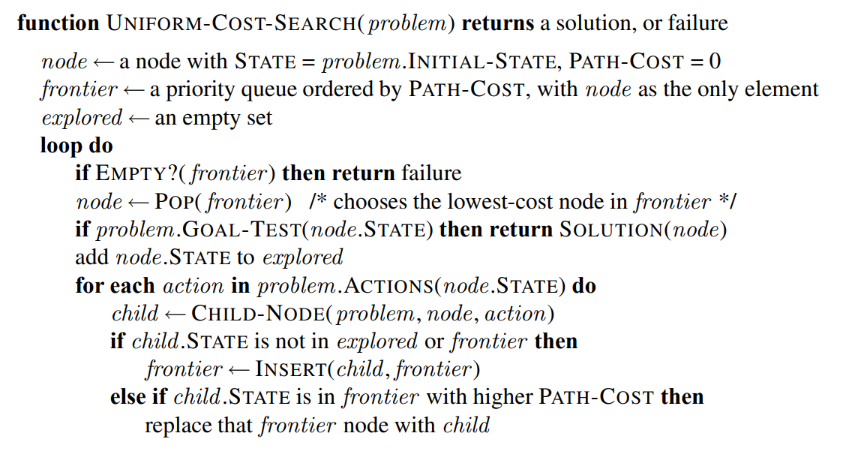
-Thuật toán cho thuật toán trên được đưa ra như sau:

+ Tạo hàng đợi ưu tiên, mảng boolean được truy cập có kích thước bằng số lượng nút và biến min\_cost được khởi tạo với giá trị tối đa. Thêm nút nguồn vào hàng đợi và đánh dấu nó đã truy cập.

+ Bật phần tử có mức độ ưu tiên cao nhất từ hàng đợi. Nếu nút bị loại bỏ là nút đích, hãy kiểm tra biến min\_cost, nếu giá trị của biến min\_cost lớn hơn chi phí hiện tại thì hãy cập nhật biến.

+ Nếu nút đã cho không phải là nút đích thì hãy thêm tất cả các nút chưa được truy cập vào hàng đợi ưu tiên liền kề với nút hiện tại.

**Thuật toán Uniform-cost search algorithm:**

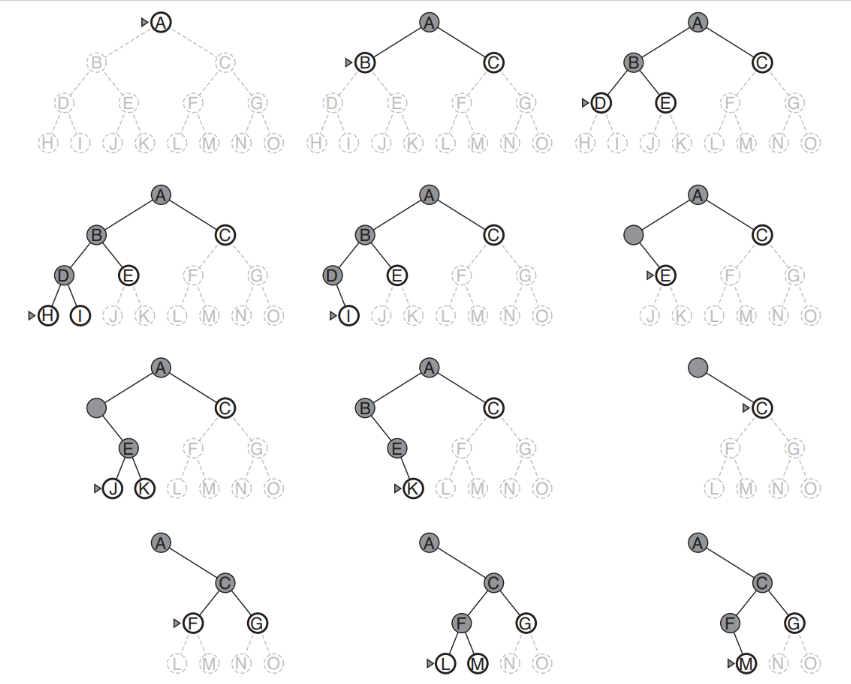


Các bài toán sửa dụng thuật toán **Uniform-cost search algorithm:**

|  |
| --- |
| **#UCS**  **def uniform\_cost\_search(problem, display=False):**  **"""[Figure 3.14]"""**  **return best\_first\_graph\_search(problem, lambda node: node.path\_cost, display)** |

**2.5.3. Depth-first search (DFS)**

- DFS là một thuật toán để duyệt hoặc tìm kiếm cấu trúc dữ liệu cây hoặc đồ thị. Thuật toán bắt đầu tại root node (chọn một số nút tùy ý làm nút gốc trong trường hợp biểu đồ) và khám phá càng xa càng tốt dọc theo mỗi nhánh trước khi quay lui. Cần thêm bộ nhớ, thường là một ngăn xếp, để theo dõi các nút được phát hiện cho đến nay dọc theo một nhánh cụ thể, giúp theo dõi lại biểu đồ. Quá trình duyệt dừng lại khi tìm thấy goal state hoặc đã duyệt hết tất cả các node.



**Thuật toán Depth-first search (DFS):**

- Giống với **Breadth-first search ( BFS)** nhưng thay vì gán frontier bằng FIFO queue thì **DEPTH-FIRST-SEARCH** gắn frontier bằng LIFO queue

**function** DEPTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure

*node* ←a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0

**if** *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

*frontier* ← a LIFO queue with *node* as the only element

*explored* ← an empty set

**loop do**

**if** EMPTY?(*frontier* ) **then return** failure

*node* ← POP(*frontier* ) /\* chooses the shallowest node in *frontier* \*/

add *node*.STATE to *explored*

**for each** *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

*child* ← CHILD-NODE(*problem, node, action*)

**if** *child*.STATE is not in( *explored* or *frontier*) **then**

if *problem*.GOAL-TEST(*child*.STATE) **then return** SOLUTION(*child*)

*frontier* ← INSERT(*child,frontier* )

Các bài toán sửa dụng thuật toán **Depth-first search (DFS)**

|  |
| --- |
| **#DFS**  **def depth\_first\_graph\_search(problem):**  **"""**  **[Figure 3.7]**  **Search the deepest nodes in the search tree first.**  **Search through the successors of a problem to find a goal.**  **The argument frontier should be an empty queue.**  **Does not get trapped by loops.**  **If two paths reach a state, only use the first one.**  **"""**  **frontier = [(Node(problem.initial))] # Stack**  **explored = set()**  **while frontier:**  **node = frontier.pop()**  **if problem.goal\_test(node.state):**  **return node**  **explored.add(node.state)**  **frontier.extend(child for child in node.expand(problem)**  **if child.state not in explored and child not in frontier)**  **return None** |

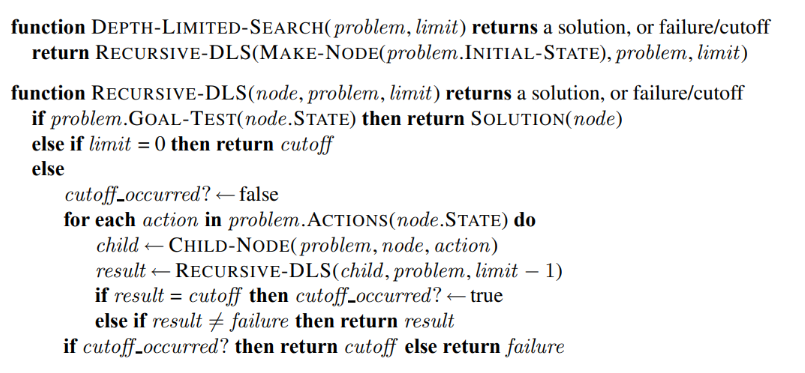
|  |  |  |
| --- | --- | --- |
|  | BFS | DFS |
| Cấu trúc dữ liệu | sử dụng cấu trúc dữ liệu Hàng đợi để tìm đường đi ngắn nhất. | sử dụng cấu trúc dữ liệu Stack. |
| Định nghĩa | BFS là một cách tiếp cận truyền tải, trong đó trước tiên chúng ta sẽ duyệt qua tất cả các nút ở cùng cấp độ trước khi chuyển sang cấp độ tiếp theo. | DFS cũng là một phương pháp truyền tải trong đó quá trình truyền tải bắt đầu từ nút gốc và tiến hành qua các nút càng xa càng tốt cho đến khi chúng ta đến được nút mà không có nút lân cận nào chưa được thăm dò. |
| Kỹ thuật | BFS có thể được sử dụng để tìm đường đi ngắn nhất của nguồn trong biểu đồ không có trọng số vì trong BFS, chúng ta đạt đến một đỉnh có số cạnh tối thiểu tính từ đỉnh nguồn. | chúng ta có thể duyệt qua nhiều cạnh hơn để đến đỉnh đích từ một nguồn. |
| Sự khác biệt về khái niệm | BFS xây dựng cây theo cấp độ. | DFS xây dựng cây con theo cây con. |
| Phương pháp tiếp cận được sử dụng | Nó hoạt động dựa trên khái niệm FIFO (Vào trước ra trước). | Nó hoạt động dựa trên khái niệm LIFO (Last In First Out). |
| Phù hợp với | BFS phù hợp hơn cho việc tìm kiếm các đỉnh gần nguồn nhất định. | DFS phù hợp hơn khi có giải pháp xa nguồn. |
| Sự phù hợp của cây quyết định | BFS xem xét tất cả các hàng xóm trước tiên và do đó không phù hợp với các cây ra quyết định được sử dụng trong trò chơi hoặc câu đố. | DFS phù hợp hơn cho các vấn đề về trò chơi hoặc câu đố. Chúng đưa ra quyết định và sau đó khám phá mọi con đường thông qua quyết định này. Và nếu quyết định này dẫn đến tình thế thắng thì chúng tôi dừng lại. |
| Độ phức tạp thời gian | Độ phức tạp thời gian của BFS là O(V + E) khi sử dụng Danh sách kề và O(V^2) khi sử dụng Ma trận kề, trong đó V là viết tắt của các đỉnh và E là viết tắt của các cạnh. | Độ phức tạp về thời gian của DFS cũng là O(V + E) khi sử dụng Danh sách kề và O(V^2) khi sử dụng Ma trận kề, trong đó V là các đỉnh và E là các cạnh. |
| Thứ tự duyệt | Duyệt theo tầng, nghĩa là tất cả các đỉnh ở cùng một mức sẽ được xem xét trước khi chuyển sang mức tiếp theo. Thường sử dụng hàng đợi để thực hiện. | Duyệt sâu vào cây, nghĩa là đỉnh gần nhánh gốc sẽ được xem xét trước. Thường sử dụng đệ quy hoặc ngăn xếp để thực hiện. |
| Loại bỏ các nút truyền tải | Các nút được duyệt qua nhiều lần sẽ bị xóa khỏi hàng đợi. | Các nút đã truy cập sẽ được thêm vào ngăn xếp và sau đó bị xóa khi không còn nút nào để truy cập. |
| Quay lui | Trong BFS không có khái niệm quay lui. | Thuật toán DFS là một thuật toán đệ quy sử dụng ý tưởng quay lui |
| Ứng dụng | đồ thị lưỡng cực, đường đi ngắn nhất, v.v. | đồ thị tuần hoàn và thứ tự tôpô, v.v. |
| Bộ nhớ | Yêu cầu nhiều bộ nhớ | Yêu cầu ít bộ nhớ |
| Sự tối ưu | BFS là tối ưu để tìm đường đi ngắn nhất. | DFS không tối ưu để tìm đường dẫn ngắn nhất |
| Độ phức tạp của không gian | độ phức tạp về không gian quan trọng hơn so với độ phức tạp về thời gian. | có độ phức tạp về không gian thấp hơn vì tại một thời điểm, nó chỉ cần lưu trữ một đường dẫn duy nhất từ nút gốc đến nút lá. |
| Tốc độ | BFS chậm so với DFS | DFS nhanh hơn so với BFS. |
| Khai thác trong vòng lặp | Trong BFS, không có vấn đề mắc kẹt trong các vòng lặp vô hạn. | Trong DFS, chúng ta có thể bị mắc kẹt trong các vòng lặp vô hạn. |
| Khi nào nên sử dụng? | Khi mục tiêu gần nguồn, BFS hoạt động tốt hơn. | Khi mục tiêu ở xa nguồn, DFS thích hợp hơn. |

**2.5.4. Depth-limited search (TREE-SEARCH version)**

**Depth-Limited Search (DLS)** là một biến thể của Depth-First Search (DFS) mà giới hạn chiều sâu của việc duyệt đỉnh trong một đồ thị. Nó giúp kiểm soát độ phức tạp của thuật toán DFS bằng cách không cho phép nó đi quá mức độ sâu được chỉ định.

Một đặc điểm chính của Depth-Limited Search là có một tham số mới, thường được gọi là "limit" hoặc "depth limit," thể hiện giới hạn về chiều sâu mà thuật toán sẽ đi vào mỗi nhánh. Nếu độ sâu hiện tại vượt quá giới hạn, thuật toán sẽ dừng lại và không tiếp tục theo nhánh đó.

**Thuật toán Depth-Limited Search (DLS):**



Các bài toán sửa dụng thuật toán **Depth-Limited Search (DLS)**

|  |
| --- |
| **def recursive\_DLS(node, problem, limit):**  **if problem.goal\_test(node.state): # Nếu vị trí đang xét là goal thì trả về node chứa kết quả**  **return node**  **elif limit == 0: # Là cột mốc đánh dấu vị trí có thể là gốc hoặc goal hoặc vị trí giới hạn của độ sâu có thể đạt tới**  **return 'cutoff'**  **else:**  **cutoff\_occurred = False**  **for child in node.expand(problem): # Mở rộng ra các node con là hậu duệ của vị trí hiện tại**  **result = recursive\_DLS(child, problem, limit - 1) # đệ quy với từng hậu duệ của vị trí hiện tại để tiếp tục tìm ra các hậu duệ tiếp theo rồi tìm ra được con đường đúng**  **if result == 'cutoff': # Kiểm tra xem đã chạm tới giới hạn của độ sâu chưa**  **cutoff\_occurred = True**  **elif result != []: # Kiểm tra xem đã chạm đến đích chưa**  **return result**  **if cutoff\_occurred:**  **return 'cutoff' # Trả về đã chạm tới giới hạn**  **else:**  **return [] # Trả về chưa tìm được đường** |

**2.6 INFORMED SEARCH ALGORITHMS**

- Là một thuật toán có them thông tin phụ trợ so với uninformed search algorithms , khi cung cấp đầu đủ thông tin bổ trợ thì nó sẽ tìm ra solution nhanh hơn nhiều lần so với uninform

|  |  |
| --- | --- |
| Uninform search | Inform search |
| - Tìm đường 1 cách ngẫu nhiên (ví dụ ở ngã 4 không biết đi đường nào)  - BFS, DFS mặc dù chọn nút sâu nhất nhưng thứ tự đưa vào tập frontier hoàn toàn ngẫu nhiên, chọn nút ban đầu hoàn toàn ngẫu nhiên.  - Không có thông tin nào cho mình biết nút này sẽ tốt hơn nút kia  - Ưu điểm chỉ cần chạy thuật toán với 5 thành phần mô tả sẽ chạy được thuật toán  - Khuyết điểm: không thể biết chính xác hướng đi nào tối ưu để tìm ra solution | - Tìm đường theo bảng chỉ dẫn (ví dụ ở ngã 4 có bảng chỉ dẫn) hướng dẫn ở trên bảng thì không chính xác tuyệt đối, chỉ là ước lượng dựa vào những thông tin đó để có thể tìm đường đi tối ưu.  - Inform search đòi hỏi mình cần phải cung cấp một thông tin nào đó bên ngoài 5 thành phần của bài toán (thông tin gợi ý )  - Thông tin gợi ý mà mình sử dụng để cài đặt thuật toán được gọi là Evaluation function f(n)  - f(n) là chi phí ước lượng tới nút n, là ước lượng toàn bộ quảng đường mình đi từ init state tới goal  - Nếu có 1 con đường đi từ init state đi qua node n thì nó được gọi là f(n) |

- Ý tưởng cho chiến lược tìm kiếm informed search

+ Thuật toán sẽ dựa vào f(n) để lựa chọn nút để đi: nếu f(n) càng nhỏ mình sẽ tin tưởng node n và lựa chọn node n

+ Đây là ý tưởng tổng quát của inform search

**A\* search**

f(n) = n \* PATH-COST + h(n)

PATH-COST là chi phí thực tế đi từ init state đến nút n của mình

h(n) : heurictic function h(n) là ước lượng chi phí từ nút n đến nút đích

Trong tập frontier thì node n có f(n) nhỏ nhất thì chọn nút đó để expand

**-** A\* là một biến thể của BFS nên nó được chứng minh là complete và optimal \* (có điều kiện)

- Ước lượng f(n) phải chính xác một cách tương đối thì A\* mới chạy đúng

- A consistent herictics satisfies

- Khi f(n) thỏa mãn tính chất sau : h(n) <= cost(n,n’) + h (n’) thì A\* optimal

|  |
| --- |
| **function** A\*-SEARCH(*problem*) **returns** a solution, or failure  *node* ←a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0  *frontier* ← a priority queue ordered by f(n), with *node* as the only element  *explored* ← an empty set  **loop do**  **if** EMPTY?(*frontier* ) **then return** failure  *node* ← POP(*frontier* ) /\* chooses the lowest-cost node in frontier \*/  **if** *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(node)  add *node*.STATE to explored  **for each** *action* **in** *problem*.ACTIONS(node.STATE) **do**  *child* ← CHILD-NODE(*problem, node, action*)  **if** *child*.STATE is not in *explored* or *frontier* **then**  *frontier* ← INSERT(*child,frontier* )  **else if** *child*.STATE is in *frontier* with higher f(n) **then**  replace that *frontier* node with *child* |

**Code demo:**

|  |
| --- |
| **# A\***  **def astar\_search(problem, h=None, display=False):**  **"""A\* search is best-first graph search with f(n) = g(n) + h(n).**  **You need to specify the h function when you call astar\_search, or**  **else in your Problem subclass."""**  **h = memoize(h or problem.h, 'h')**  **return best\_first\_graph\_search(problem, lambda n: n.path\_cost + h(n), display)** |

Lưu ý A\* được coi là một biến thể của bfs do đó nó có ưu điểm và khuyết điểm của bfs

Ưu điểm Complete Optimal \*

Nhược điểm Space Time

Để cái tiến thì dùng thuật toán SMA\*

- Xây dựng h(n)

+ Min h(n) = 0

+ H(n) <= chi phí thực tế actual cost

+Trong một bài toán thì có thể có nhiều hàm h(n) -> ưu tiên chọn hàm h(n) lớn hơn

- Ví dụ: Xây dung h(n) cho bài toán 8 -puzzle

+ h1 là số lượng các ô đặt sai vị trí ở 1 state so với goal

+ h2 là tổng khoảng cách từ 1 ô từ vị trí hiện tại đến vị trí đích đến của nó và ô này không di chuyển theo chiều chéo

- Xây dựng heuristic

theo Relax Problems : làm nhẹ vấn đề, làm giảm yêu cầu của bài toán

Ví dụ bài toán 8 -puzzle

- Ta có thể di chuyển một ô từ A đến ô B nếu A và B liền kề với nhau và ô B phải trống

+ Relax Problem A: ô A có thể di chuyển sang ô B nếu A và B cạnh nhau

+ Relax Problem B: ô A có thể di chuyển sang ô B nếu B trống

+ Relax Problem C: ô A có thể di chuyển sang ô B

Cost problem A,B,C = h(n)

H1->A

H2->B

H2->C

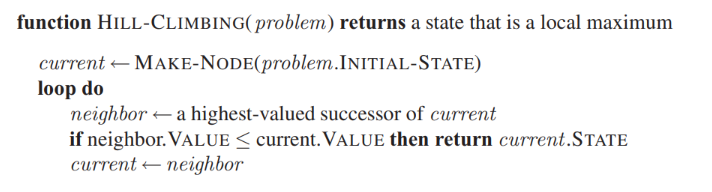
H(n) = max(h1,h2,h3)

**2.7 LOCAL SEARCH - SEARCHING FOR GOAL STATES**

- Thuật toán Local search là thuật toán tìm kiếm cục bộ là một loại thuật toán tối ưu hóa hoạt động trên một trạng thái hiện tại và lặp đi lặp lại chuyển đến một trạng thái láng giềng cải thiện hàm mục tiêu. Khác với thuật toán tìm kiếm toàn cục khám phá toàn bộ không gian tìm kiếm, tìm kiếm cục bộ tập trung vào khu vực xung quanh trạng thái hiện tại. Khi tìm kiếm trạng thái mục tiêu, các thuật toán tìm kiếm cục bộ nhằm tìm một trạng thái thỏa mãn điều kiện mục tiêu.

**2.7.1 Hill-climbing search**

**Thuật toán Hill-climbing search:**



**Code demo:**

|  |
| --- |
| **# Hill climbing**  **def hill\_climbing(problem):**  **current = Node(problem.initial)**  **current\_values = manhattan\_distance(current.state, problem.goal)**    **best\_neighbor = choose\_best\_neighbor(problem, current)**  **while manhattan\_distance(best\_neighbor.state, problem.goal) < current\_values:**  **if problem.goal\_test(best\_neighbor.state):**  **return best\_neighbor**    **current = best\_neighbor**  **current\_values = manhattan\_distance(best\_neighbor.state, problem.goal)**  **best\_neighbor = choose\_best\_neighbor(problem, current)**  **return best\_neighbor**  **def choose\_best\_neighbor(problem, current):**  **neighbors = current.expand(problem)**  **best\_neighbor = neighbors[0]**  **for i in range(0, len(neighbors), 1):**  **if manhattan\_distance(neighbors[i].state, problem.goal) < manhattan\_distance(best\_neighbor.state, problem.goal):**  **best\_neighbor = neighbors[i]**  **return best\_neighbor**  **def manhattan\_distance(state, goal\_state):**  **# Calculate the Manhattan distance heuristic**  **total\_distance = 0**  **for i in range(len(state)):**  **if state[i] == 0:**  **continue**  **for j in range(len(state)):**  **if state[i] == goal\_state[j]:**  **total\_distance += abs(j - i)**    **return total\_distance** |

- Thuật toán Hill-Climbing là một thuật toán tìm kiếm cục bộ, trong đó ý tưởng chính là luôn chọn trạng thái láng giềng tốt nhất xung quanh trạng thái hiện tại để di chuyển tới. Tuy nhiên, thuật toán này có nhược điểm là chỉ đảm bảo tìm được local maximum, không phải global maximum, và có thể bị kẹt tại các điểm tối ưu cục bộ (local maximum) hoặc trên các mặt phẳng bằng ngang (plateaux).

- Có hai vấn đề chính với Hill-Climbing search:

+ Quá Nhiều "Successors": Thuật toán có thể tạo ra quá nhiều trạng thái láng giềng, làm tăng độ phức tạp của quá trình tìm kiếm.

+ Kẹt Ở Local Maximum hoặc Plateaux: Hill-Climbing có thể dễ dàng bị kẹt tại các điểm tối ưu cục bộ hoặc trên các mặt phẳng bằng ngang, không thể tiếp tục tìm kiếm global maximum.

+ Để giải quyết vấn đề quá nhiều "successors," có một phiên bản cải tiến của Hill-Climbing được gọi là Stochastic Hill Climbing. Thuật toán này tạo ra một trạng thái láng giềng ngẫu nhiên và chấp nhận nó nếu cải thiện hàm mục tiêu, ngược lại sẽ tạo ra một trạng thái láng giềng khác.

+ Để vượt qua vấn đề kẹt ở local maximum hoặc plateaux, có các phương pháp sau:

+ Cho phép Di Chuyển Ngang (Sideways Moves): Hill-Climbing có thể thực hiện các bước di chuyển ngang để thoát khỏi plateaux. Tuy nhiên, cần giới hạn số lần di chuyển ngang để tránh vòng lặp vô tận.

+ Random-Restart Hill Climbing: Chạy lại thuật toán nhiều lần với các trạng thái khởi tạo ngẫu nhiên để tìm kiếm từ nhiều điểm xuất phát khác nhau.

+ Local Beam Search: Thay vì thực hiện tìm kiếm ngẫu nhiên, chạy nhiều tìm kiếm cùng một lúc và chọn ra trạng thái tốt nhất.

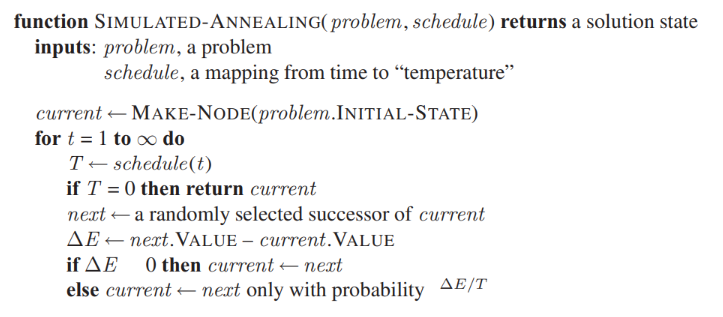
+ Stochastic Beam Search: Kết hợp giữa local beam search và stochastic hill climbing, chọn trạng thái không chỉ dựa trên giá trị tốt nhất mà còn dựa trên xác suất.

+ Simulated Annealing: Mô phỏng quá trình rèn kim loại, thuật toán này cho phép những bước di chuyển xấu hơn theo một xác suất giảm dần, giúp thoát khỏi local maximum.

**2.7.2 Simulated annealing algorithm**

- Thuật toán Simulated Annealing là một thuật toán tìm kiếm cục bộ được mô phỏng theo quá trình rèn kim loại (annealing) trong lĩnh vực vật liệu. Ý tưởng chính của thuật toán là cho phép các bước di chuyển xấu hơn theo một xác suất giảm dần, giúp thoát khỏi các điểm tối ưu cục bộ và có khả năng tiếp tục tìm kiếm global optimum.

**Thuật toán Simulated annealing algorithm:**



**Thuật toán demo:**

|  |
| --- |
| **''' WRITE THIS FUNCTION: '''**  **def simulated\_annealing(problem):**  **"""See [Figure 4.5] for the algorithm."""**  **current = Node(problem.initial)**  **for t in range(1, sys.maxsize):**  **#Schedule(t) để tính thời gian chuyển động**  **T = schedule(t)**  **# Hết thời gian chuyển động => nhiệt độ = 0 => dừng**  **if T == 0:**  **return current.state**  **# Lấy các successor có thể di chuyển từ vị trí đang đứng**  **successor = current.expand(problem)**  **# Kiểm tra xem tập successor có rỗng không**  **if successor != []:**  **# Lấy ngẫu nhiên một successor trong danh sách có thể đi**  **next\_state = random.choice(successor)**  **# Kiểm tra xem vị trí ngẫu nhiên này có bị xung đột với các con hậu đang có trên bàn không**  **deltaE = problem.value(next\_state) - problem.value(current)**  **# Nếu không xung đột hoặc xung đột trong phạm vị cho phép thì gán vị trí mới thành vị trí hiện tại và cho con hậu lên bàn cờ**  **if deltaE > 0 or random.random() < np.exp(-deltaE / T):**  **current = next\_state**  **# Nếu không tìm ra thì quay lại vòng lặp tìm tiếp**  **else:**  **return current.state** |

**2.8. Searching in nondeterministic environments**

Môi trường là hoàn toàn quan sát và xác định, và tác nhân biết rõ ảnh hưởng của mỗi hành động. Do đó, tác nhân có thể dự đoán chính xác trạng thái kết quả của mọi chuỗi hành động và luôn hiểu rõ tình trạng hiện tại của mình. Tuy nhiên, thông tin từ cảm nhận sau mỗi hành động không mang lại điều gì mới, chỉ giúp tác nhân biết trạng thái khởi đầu.

Khi môi trường chỉ là một phần quan sát được hoặc không xác định (hoặc cả hai), cảm nhận trở nên quan trọng. Trong môi trường một phần quan sát được, mỗi cảm nhận giúp giới hạn các trạng thái có thể tác nhân đang ở, làm cho việc đạt được mục tiêu trở nên dễ dàng hơn. Khi môi trường không xác định, cảm nhận chỉ ra kết quả cụ thể của hành động của tác nhân trong số nhiều kết quả có thể xảy ra. Cả hai trường hợp đều khiến cho tương lai của cảm nhận không thể được dự đoán trước và hành động tương lai của tác nhân sẽ phụ thuộc vào những cảm nhận đó.

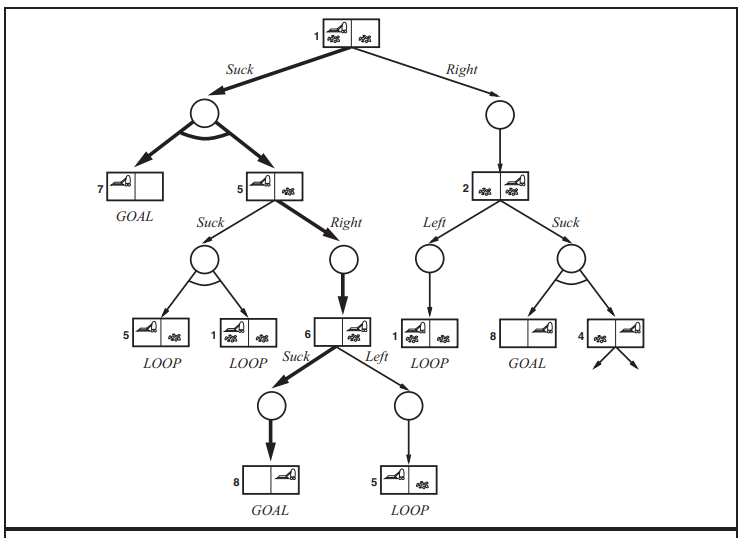
Trong trường hợp môi trường không xác định, giải pháp không phải là một chuỗi hành động mà là một kế hoạch có điều kiện, còn được gọi là chiến lược có điều kiện. Thuật toán này chỉ định hành động dựa trên các cảm nhận nhận được.

**2.8.1. AND-OR search trees**

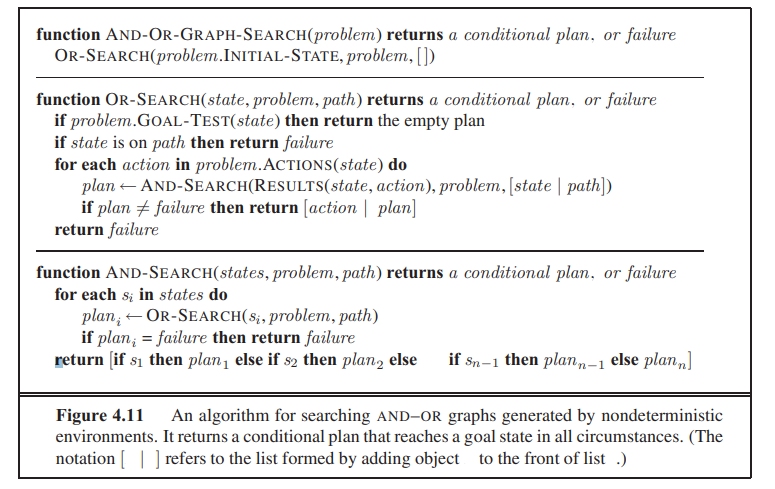
Thuật toán AND-OR Search Trees thường được sử dụng để mô hình hóa và giải quyết các vấn đề quyết định trong môi trường không xác định, đặc biệt là trong ngữ cảnh của cây quyết định.

OR Nodes: Đây là những node đại diện cho các trạng thái, tương tự như trong tìm kiếm thông thường. Mỗi OR node kết nối đến các trạng thái kế tiếp mà một hành động có thể dẫn đến.

AND Nodes: Đây là các node đặc biệt và có thể được hiểu như là đường dẫn tới tất cả các trạng thái kế tiếp có thể xảy ra từ một OR node. AND node đại diện cho sự không chắc chắn về việc hành động sẽ dẫn đến trạng thái cụ thể nào.



Các chiến lược tìm kiếm được áp dụng ở mỗi nút AND hoặc OR để quyết định hành động nào nên được chọn tiếp theo. Thuật toán Minimax thường được tích hợp để đưa ra quyết định tối ưu trong trò chơi và các tình huống đối đầu.



**Code demo:**

|  |
| --- |
| **''' IMPLEMENT THE FOLLOWING FUNCTION '''**  **def and\_or\_graph\_search(problem):**  **"""See [Figure 4.11] for the algorithm"""**  **return or\_search(problem.initial, problem, [])**  **def or\_search(state, problem, path):**  **# Kiem tra xem da tim ra được solution chưa**  **if problem.goal\_test(state):**  **return []**  **# Nếu vị trí hiện tại đã được duyệt qua thì hủy vị trí này**  **if state in path:**  **return None**  **# Duyệt ra từng action tiếp theo có thể đi từ node hiện tại**  **for action in problem.actions(state):**  **# Dừng bộ AND\_SEARCH để chỉ ra địa chỉ tiếp theo**  **plan = and\_search(problem.result(state, action), problem, path + [state, ])**  **# Nếu có địa chỉ tiếp theo thì trả về action và hành đi động di chuyển**  **if plan != None:**  **return [action, plan]**  **return None**  **def and\_search(states, problem, path):**  **plan = {}**  **# Đệ quy để chỉ ra các node tiếp theo**  **plan[states] = or\_search(states, problem, path)**  **if plan[states] == None:**  **return None**  **# Trả về kế hoạch đã được duyệt qua**  **return plan** |

**Function** AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure

Mục đích: Trả về một kế hoạch có điều kiện hoặc thất bại.

Phương pháp: Gọi OR-SEARCH với trạng thái ban đầu và trả về kết quả.

**Function** OR-SEARCH(state, problem, path) returns a conditional plan, or failure

Mục đích: Trả về một kế hoạch có điều kiện hoặc thất bại dựa trên trạng thái hiện tại.

Phương pháp: Kiểm tra xem trạng thái hiện tại có phải là trạng thái mục tiêu không. Kiểm tra xem trạng thái có trên đường dẫn không (để tránh chu trình).

Duyệt qua các hành động có thể thực hiện tại trạng thái và gọi AND-SEARCH với các trạng thái kế tiếp được tạo ra từ mỗi hành động. Nếu có kế hoạch từ AND-SEARCH không thất bại, thì thêm hành động hiện tại vào kế hoạch đó và trả về.

**Function** AND-SEARCH(states, problem, path) returns a conditional plan, or failure

Mục đích: Trả về một kế hoạch có điều kiện hoặc thất bại dựa trên tập hợp các trạng thái.

Phương pháp: Duyệt qua từng trạng thái trong tập hợp. Gọi OR-SEARCH cho mỗi trạng thái, kiểm tra xem có kế hoạch không. Nếu có ít nhất một kế hoạch không thất bại, trả về kế hoạch đầu tiên không thất bại. Nếu tất cả đều thất bại, trả về thất bại.

Tổ chức của Kế Hoạch Có Điều Kiện:

Kế hoạch có điều kiện được tổ chức dưới dạng một cây quyết định.

Mỗi nút lá của cây đại diện cho một hành động, và mỗi nút nội địa của cây đại diện cho một trạng thái có nhiều hơn một lựa chọn.

Các Dòng Lệnh Cụ Thể:

Các dòng lệnh cuối cùng của mã nguồn trả về kết quả của AND-SEARCH dưới dạng một kế hoạch có điều kiện, được tổ chức theo cây quyết định.

Tổng quan, mã nguồn này mô phỏng một quy trình tìm kiếm trong không gian trạng thái không xác định và xây dựng kế hoạch có điều kiện dựa trên các quyết định ở cấp độ trạng thái và hành động.

**2.9. Search in partially observable environments**

Nondeterministic Environments: trong môi trường không xác định, mỗi hành động không đưa ra một kết quả cụ thể, mà có thể dẫn đến nhiều trạng thái khác nhau. Điều này phản ánh sự không chắc chắn về kết quả của hành động trong môi trường.

**2.9.1. Searching with no observation**

Trong trường hợp "Sensorless" (tức là không có cảm biến), con agent không có thông tin nào về môi trường. Điều này có thể xảy ra khi việc trang bị cảm biến là không khả thi hoặc không hiệu quả.

Agent loại này giải quyết vấn đề của nó ít hiệu quả hoặc thậm chí không giải quyết được nếu không có thông tin về môi trường.

Ý tưởng để giải quyết bài toán này là xây dựng "belief state" cho con agent, một tập hợp các trạng thái vật lý mà con agent có thể đang ở, dựa trên quan sát và kiến thức trước đó.

Belief state là một tập hợp các trạng thái vật lý có thể của con agent tại vị trí nó đang đứng. Nó đại diện cho sự không chắc chắn về trạng thái hiện tại và được sử dụng để đưa ra quyết định. Sau khi có belief state, agent có thể sử dụng các thuật toán fully observable để thực hiện quyết định và hành động trong môi trường không hoàn toàn quan sát.

**2.9.2. Searching for partially observable problems**

Trong bài toán tìm kiếm trong môi trường một phần quan sát (PARTIALLY OBSERVABLE PROBLEMS), khi có một số lượng nhỏ cảm biến, quá trình này thường bao gồm ba thành phần chính: State Prediction, Observation, và State Update.

1. *State Prediction (Dự Đoán Trạng Thái)*

Trước khi nhận được quan sát mới từ cảm biến, agent cần dự đoán trạng thái tiếp theo dựa trên thông tin hiện có. Điều này thường thực hiện thông qua một mô hình dự đoán trạng thái, có thể là một mô hình xác suất dựa trên lịch sử các trạng thái và hành động trước đó. Mục tiêu là tạo ra một dự đoán có thể sử dụng cho quyết định tiếp theo.

1. *Observation (Quan Sát)*

Sau khi đã có dự đoán trạng thái, agent nhận được thông tin từ cảm biến, tạo thành một quan sát mới về môi trường. Quan sát này có thể không chắc chắn và thường đi kèm với mức độ nhiễu, phản tác động của môi trường, hoặc các yếu tố khác có thể làm cho thông tin trở nên không chắc chắn.

1. *State Update (Cập Nhật Trạng Thái)*

Sau khi nhận được quan sát mới, agent cần cập nhật trạng thái của mình dựa trên thông tin mới này. Quá trình này thường sử dụng mô hình cập nhật trạng thái, có thể dựa trên lý thuyết xác suất. Mục tiêu là làm cho trạng thái của agent trở nên chính xác hơn và phản ánh thông tin mới nhất từ quan sát.

Quy trình tổng cộng này giúp agent xây dựng và duy trì niềm tin về trạng thái của môi trường, gọi là "belief state." Belief state này không chỉ là một trạng thái duy nhất mà là một phân phối xác suất trên các trạng thái có thể có, dựa trên dữ liệu quan sát và dự đoán trước đó.

Các bước trên giúp agent nắm bắt được thông tin từ môi trường một cách linh hoạt và làm cho quá trình ra quyết định trở nên hiệu quả hơn trong môi trường không hoàn toàn quan sát.

**2.10. Searching with no transition model - online search**

Online search sẽ hữu ích trong một số trường hợp như : Khi không có mô hình của môi trường, Khi agent đứng lâu sẽ bị chịu phát (ví dụ dây chuyền sản xuất,….) , môi trường nondeterministic,…

Phép đo dùng để đánh giá độ hiệu quả của thuật toán

Competitive Ratio = Chi phí thực tế (Actual path cost) / Chi phí đường đi ngắn nhất

Giá trị của Competitive Ratio >= 1

Dead-end state là một action nếu nó không thể nào quay ngược lại. Môi trường không có dead-end state được gọi là safely exploreable

**2.10.1 Online depth-first search agent:**

Một agent tìm kiếm online depth-first search là một thuật toán tìm kiếm theo chiều sâu có khả năng hoạt động trong môi trường thực tế. Trong ngữ cảnh của môi trường thực tế, agent xử lý thông tin một cách từng phần, điều chỉnh cho dữ liệu mới khi nó trở nên khả dụng.

Mỗi một bước chạy (lần đi agent), sẽ quan sát được s’ state của môi trường, vậy thì đứng ở mỗi trạng thái state ta nên có action gì thì thuật toán sau sẽ giúp ta làm điều đó

**Thuật toán Online depth-first search agent**

A screenshot of a computer code

Description automatically generated

- Lý do sử dụng Depth-first vì agent đang chạy trực tiếp trong môi trường. Khi muốn quay lại vị trí trước đó thì chỉ cần tốn 1 chi phí để có thể quay lại. Nếu sử dụng bfs thì chi phí để quay lại rất lớn.

**2.10.2: Online A\* search LRTA A\* (Learning Real-Time A\*) algorithm**

Cũng là thuật toán online agent nên kết quả cũng trả về là action, cho vào s’ : cho mình biết state hiện tại đang đứng, mục đích không chỉ tạo ra action mà còn tạo ra result, thuật toán này không sử dụng unbacktracked và untried giống depth first mà tạo ra bảng H: ước lượng quãng đường đi từ trạng thái s nào đó cho đến đích, nếu đạt đến đích thì dừng lại

Ý tưởng: Tại mỗi bước của state s, thì tập frontier bị giới hạn còn lại những successor’s với chi phì f(n) là tối ưu.

f(s) = path-cost(s) \* h(s) => f(s) = cost(s,s’) + h(s’)

H(s) cập nhật h(s) thành H(s)

**Thuật toán Online A\* search LRTA A\* (Learning Real-Time A\*) algorithm**

A white paper with black text

Description automatically generated

**2.11. Solve CSPs using Constraint Propagation**

Các thành phần của bài toán csps

Variable

Domains

Constraint

Giải bài toán csp là tìm các giá trị để điền bào biến X ở variblale

using Constrait Propagation giải những bài toán csp đơn giản

phương pháp này có thể ko giải được nhưng có thể thu hẹp domain lại

Khi thu hẹp domain thì nó có thể tự động thỏa mãn một số constraint

Những biến liên quan đến domain bị thu hẹp thì domain cũng có thể bị thu hẹp

Tới một lúc nào thì domain sẽ thu hẹp về còn 1 giá trị, lúc đó biến x sẽ lưu -> nhiều lần như vậy thì có thể giải được bài toán cho tất cả các biến

Nhưng có trường hợp domain còn lại nhiều biến, lúc đó bài toán này sẽ không thể dùng thuật toán này để giải được

Constraint graph

Một variable là một node. Nếu 2 biến có tồn tại constraint thì sẽ tồn tại đường nối với nhau

Consitency là một tập hợp mà lấy bất kỳ giá trị nào trong domain của nó thì cũng thỏa mãn constraint

* Thuật toán này thúc đẩy local consistency bằng cách thu nhỏ domain của nó bằng cách loại bỏ các giá trị vi phạm constraint

Một số loại Consistency :

Node consistency là những consistency trong nội bộ cái node, một node là một biến, thúc đẩy constraint chứa một biến

Arc consistency là consistency 2 biến

Path consistency là những constraint >= 3 biến

Node consistency đảm bảo mọi constraint đều là một biến

Một đồ thị constraint graph đạt đến trạng thái node consistency nếu mọi constraint một biến đều thỏa

* Khi đạt trạng thái node consistency thì có thể xóa bỏ hết tất cả constraint một biến

Arc consistency là consistency 2 biến

Việc thu hẹp domain cho arc consistency là một phép toán có thứ tự. Trong trường hợp khổng thể thu hẹp domain của biến thì cần sử dụng các phương pháp mạnh hơn

Thuật toán cài đặt AC-3

A screenshot of a white sheet with black text

Description automatically generated

**Code demo:**

|  |
| --- |
| **def AC3(csp):**  **"""See [Figure 6.3] for the algorithm"""**  **queue = {(Xi, Xk) for Xi in csp.variables for Xk in csp.neighbors[Xi]}**  **csp.curr\_domains = csp.domains.copy() # curr\_domains: a copy of domains. We will do inference on curr\_domains**    **''' ADD YOUR CODE HERE '''**  **while queue:**  **# Lấy từng cặp constrains ra để kiểm tra**  **(Xi, Xj) = queue.pop()**  **# Kiểm tra xem có cặp nào bị thay đổi không**  **if revise(csp, Xi, Xj):**  **# Nếu không có cặp nào được sinh ra thì constrains không thỏa**  **if not csp.curr\_domains[Xi]:**  **return False**  **# Thêm các cặp mới vào**  **for Xk in csp.neighbors[Xi] - {Xj}:**  **queue.add((Xk, Xi))**  **return True # CSP is satisfiable**  **def revise(csp, Xi, Xj):**  **"""Return true if we remove a value."""**  **revised = False**  **for x in csp.domains[Xi]:**  **conflict = True**    **for y in csp.curr\_domains[Xj]:**  **if csp.constraints(Xi, x, Xj, y):**  **conflict = False**  **break**  **if conflict:**  **csp.domains[Xi].remove(x)**  **revised = True**    **return revised** |

Path Consistency

Có 1 vài trường hợp thì arc consistency không thể thu hẹp domain vì vậy ta cần tăng số biến cần kiểm tra lên chăng hạng 3 biến là path consistency. Thuật toán PC-2

K – Consistency

Là phiên bản tổng quát của consistency. K = 3 -> path consistency

Ở đây ta có K biến ta sẽ lấy k – 1 biến có giá trị và tím giá trị cho biến thứ k

Lưu ý :

Khi dùng phương pháp Consistency ở K = 3 vẫn không thể giải được bài toán thì ta cần dùng một phương pháp mạnh hơn.

**2.12.** **Backtracking Search Algorithm**

Backtracking Search Algorithm là một phương pháp hiệu quả để giải quyết các vấn đề thuộc lĩnh vực Ràng buộc (Constraint Satisfaction Problems - csps). Dưới đây là mô tả chi tiết về cách thuật toán hoạt động và cách nó sử dụng các thành phần của csps

Quy Trình Cơ Bản:

Chọn Biến: lựa chọn một biến chưa được gán giá trị từ tập hợp các biến (X).

Thử Giá Trị: chọn một giá trị từ miền giá trị của biến đã chọn.

Kiểm Tra Consistency: kiểm tra xem giá trị đã chọn có consistent với assignment (tập hợp các giá trị đã gán cho các biến khác) không. Consistency ở đây thường bao gồm việc đảm bảo rằng không có hai biến nào có giá trị trùng nhau và không vi phạm các ràng buộc.

Lựa Chọn Tiếp Theo: nếu giá trị chọn là consistent, chọn một biến tiếp theo và thử giá trị cho nó.Quá trình này được lặp lại cho đến khi tìm được một giải pháp hoặc không còn giá trị nào để thử.

Backtrack Nếu Cần: nếu không tìm được giá trị phù hợp cho biến, quay lại bước trước đó (backtrack) và thử giá trị khác cho biến trước đó.

Kiểm Tra Kết Thúc: kiểm tra xem đã có giải pháp chưa. Nếu tìm được giải pháp, dừng lại. Ngược lại, quay lại bước trước đó và thử lại.

Sử Dụng CSP Components (X, D, C):

Biến (X): là tập hợp các biến cần gán giá trị. Mỗi biến đại diện cho một phần của vấn đề.

Miền Giá Trị (D): là tập hợp các giá trị mà mỗi biến trong X có thể nhận được.

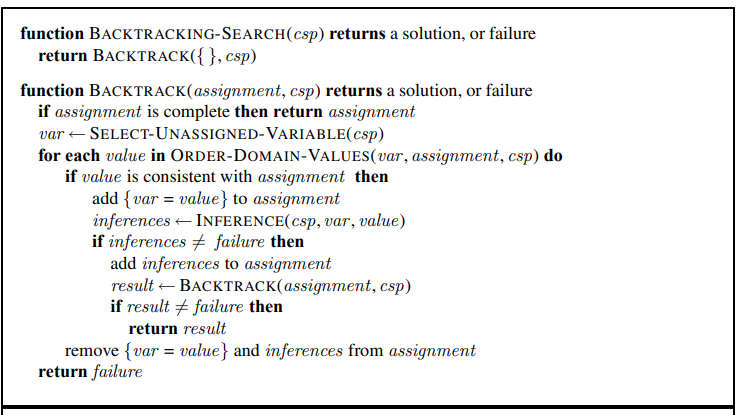
Ràng Buộc (C): là tập hợp các ràng buộc mà giữa các biến phải tuân theo. Ràng buộc này xác định các điều kiện giới hạn về cách các biến có thể kết hợp.

Mở Rộng Ý Tưởng:

Heuristics: sử dụng các chiến lược heuristics để chọn biến và giá trị một cách thông minh, giảm độ phức tạp của quá trình tìm kiếm.

Forward Checking: kiểm tra sự hợp lệ của giá trị ngay sau khi nó được gán cho biến, giúp dẫn đến việc lựa chọn hiệu quả hơn.

Constraint Propagation: tích hợp các phương pháp constraint propagation để nhanh chóng loại bỏ các giá trị không thể và tăng hiệu suất của thuật toán.



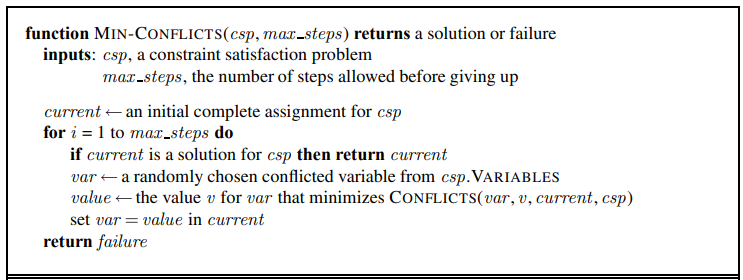
**Code demo:**

|  |
| --- |
| **#%% CSP Backtracking Search**  **# Variable ordering**  **def first\_unassigned\_variable(assignment, csp): #random selection**  **"""The default variable order."""**  **return first([var for var in csp.variables if var not in assignment])**  **def num\_legal\_values(csp, var, assignment):**  **if csp.curr\_domains:**  **return len(csp.curr\_domains[var])**  **else:**  **return count(csp.nconflicts(var, val, assignment) == 0 for val in csp.domains[var])**  **def minimum\_remaining\_values(assignment, csp):**  **"""Minimum-remaining-values heuristic."""**  **return argmin\_random\_tie([v for v in csp.variables if v not in assignment],**  **key=lambda var: num\_legal\_values(csp, var, assignment))**  **# Value ordering**  **def unordered\_domain\_values(var, assignment, csp): #random selection**  **"""The default value order."""**  **return (csp.curr\_domains or csp.domains)[var]**  **def least\_constraining\_value(var, assignment, csp):**  **"""Least-constraining-values heuristic."""**  **return sorted((csp.curr\_domains or csp.domains)[var], key=lambda val: csp.nconflicts(var, val, assignment))**  **# Inference**  **def forward\_checking(csp, var, value, assignment, removals):**  **"""Prune neighbor values inconsistent with var=value."""**  **for B in csp.neighbors[var]:**  **if B not in assignment:**  **for b in csp.curr\_domains[B][:]:**  **if not csp.constraints(var, value, B, b):**  **csp.curr\_domains[B].remove(b)**  **if removals is not None:**  **removals.append((B, b)) # variable B and value b are removed from its domain**  **if not csp.curr\_domains[B]:**  **return False**  **return True**    **# Backtracking search**  **def backtracking\_search(csp, select\_unassigned\_variable=minimum\_remaining\_values,**  **order\_domain\_values=least\_constraining\_value,**  **inference=forward\_checking):**  **def backtrack(assignment):**  **return False**    **return False**    **def restore(csp, removals):**  **"""Undo a supposition and all inferences from it."""**  **for B, b in removals:**  **csp.curr\_domains[B].append(b)** |

* If assignment is complete then return assignment: Nếu phân công đã hoàn chỉnh (đã gán giá trị cho tất cả các biến), trả về phân công đó làm giải pháp.
* Var SELECT-UNASSIGNED-VARIABLE(csp): Chọn một biến chưa được gán giá trị từ csp.
* For each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do: Duyệt qua các giá trị từ tập giá trị của biến được chọn, theo thứ tự được xác định bởi hàm ORDER-DOMAIN-VALUES.
* If value is consistent with assignment then: Nếu giá trị đó nhất quán với phân công hiện tại.
* Add {var = value} to assignment: Thêm {var = value} vào phân công.
* Inferences INFERENCE(csp, var, value): Tạo các suy luận dựa trên việc gán giá trị và kiểm tra ràng buộc.
* If inferences ≠ failure then: Nếu suy luận không thất bại.
* Add inferences to assignment: Thêm suy luận vào phân công.
* Result BACKTRACK(assignment, csp): Gọi đệ quy hàm BACKTRACK với phân công và csp.
* If result ≠ failure then: Nếu kết quả của đệ quy không thất bại.
* Return result: Trả về kết quả đó làm giải pháp.
* Remove {var = value} and inferences from assignment: Loại bỏ {var = value} và suy luận từ phân công.
* Return failure: Trả về thất bại nếu không tìm thấy giải pháp.

**2.13. Local Search for CSPs**

Tìm kiếm cục bộ cho Bài toán Ràng buộc (Local Search for CSPs) là một phương pháp giải quyết bài toán ràng buộc bằng cách tập trung vào việc cải thiện từng bước tại một trạng thái hiện tại, thay vì xem xét toàn bộ không gian trạng thái như các thuật toán tìm kiếm thông tin. Ở mỗi bước của thuật toán, một trạng thái mới được tạo ra bằng cách thay đổi giá trị của một biến nào đó trong trạng thái hiện tại. Mục tiêu là di chuyển từ trạng thái hiện tại đến một trạng thái mới, mỗi lần cải thiện hoặc duy trì sự đồng thuận với các ràng buộc. Thuật toán tiếp tục lặp lại quá trình này cho đến khi đạt được một trạng thái mà không thể cải thiện được nữa hoặc đạt đến một điều kiện dừng được xác định. Một trong những heuristics phổ biến được sử dụng trong tìm kiếm cục bộ cho CSPs là "MIN-CONFLICTS." Nó chọn giá trị mới cho biến hiện tại sao cho nó tạo ra ít xung đột nhất với các biến khác. Các xung đột ở đây đề cập đến việc vi phạm ràng buộc trong trạng thái hiện tại. Tìm kiếm cục bộ thường được ưa chuộng trong các bài toán ràng buộc lớn, nơi không gian trạng thái lớn và việc kiểm tra toàn bộ không gian trạng thái là không hiệu quả. Thay vào đó, nó tập trung vào việc cải thiện từng trạng thái hiện tại một cách linh hoạt và hiệu quả.



**Code demo:**

|  |
| --- |
| **#%% Min-conflicts for CSPs**  **''' READ AND COMMENT to show your comprehensive understanding of the following function '''**  **def min\_conflicts(csp, max\_steps=100000):**  **"""See Figure 6.8 for the algorithm"""**  **# Gán giá trị hiện tại cho biến current của CSP**  **csp.current = current = {}**  **# Lọc từng giá trị trong domains**  **for var in csp.variables:**  **# Chọn giá trị có ít xung đột nhất cho biến và gán nó**  **val = min\_conflicts\_value(csp, var, current)**  **csp.assign(var, val, current)**    **# Bắt đầu vòng lặp chính**  **for i in range(max\_steps):**  **# Kiểm tra biến xung đột**  **conflicted = csp.conflicted\_vars(current)**  **# Nếu không có biến xung đột, trả về biến hiện tại**  **if not conflicted:**  **return current**  **# Ngẫu nhiên chọn một biến xung đột**  **var = random.choice(conflicted)**  **# Xét biến ít xung đột nhất**  **val = min\_conflicts\_value(csp, var, current)**  **# Đăng kí lại giá trị đã dùng để không bị dính constrain**  **csp.assign(var, val, current)**  **return None**  **def min\_conflicts\_value(csp, var, current):**  **"""Return the value that will give var the least number of conflicts.**  **If there is a tie, choose at random."""**  **return argmin\_random\_tie(csp.domains[var], key=lambda val: csp.nconflicts(var, val, current))** |

* **function** MIN-CONFLICTS(csp, max\_steps) **returns** a solution or failure: đây là khai báo của hàm, nó nhận vào hai đối số là bài toán ràng buộc (csp) và số bước tối đa được phép thực hiện trước khi từ bỏ (max\_steps). Hàm trả về một giải pháp hoặc thất bại.
* **inputs:** csp, a constraint satisfaction problem max\_steps, the number of steps allowed before giving up: Đây là mô tả cho đầu vào của hàm, gồm bài toán ràng buộc (csp) và số bước tối đa (max\_steps).
* current an initial complete assignment for csp: current là một gán giá trị ban đầu hoàn chỉnh cho bài toán ràng buộc.
* **for** i = 1 to max\_steps **do**: Bắt đầu vòng lặp, thực hiện max\_steps lần.
* **if** current is a solution for csp **then return** current: Nếu current là một giải pháp cho bài toán ràng buộc (csp), thì trả về current làm kết quả.
* var - a randomly chosen conflicted variable from csp: Chọn một biến có xung đột ngẫu nhiên từ bài toán ràng buộc (csp).
* VARIABLES value the value v for var that minimizes CONFLICTS(var, v, current, csp): Chọn giá trị (value) cho biến (var) sao cho giảm thiểu số lượng xung đột khi gán giá trị này.
* set var = value in current: Gán giá trị mới (value) cho biến (var) trong current.
* **return failure:** Trả về thất bại nếu không tìm thấy giải pháp sau số lượng bước tối đa được quy định.

PHẦN 3: LẬP TRÌNH

## **LẬP TRÌNH GIAO DIỆN GAME 8 - PUZZLE.**

### Giới thiệu chung về chủ đề

#### Mô tả yêu cầu của chủ đề

+ Lập trình Game 8 - puzzle bằng thư viện giao diện tinker

+ Gồm 3 phần (Part) cơ bản sau

. Thư mục chứa thuật toán, các class hỗ trợ khai báo từng đối tượng cụ thể, các module hỗ trợ chạy thuật toán:

. Hiển thị giao diện cho người chơi.

. Tiến hành chạy từng thuật toán để giải quyết vấn đề.

#### Phân tích yều cầu của chủ đề

+ Chủ đề lập trình python này là cơ sở của các ứng dụng về AI, ML, Deep ML và tập trung nhiều vào các thuật toán.

+ Ứng dụng các thuật toán thường dùng trong AI này vào giải quyết vấn đề cụ thể qua từng ma trận khác nhau.

### Các nền tảng kỹ thuật

#### Nền tảng kỹ thuật cơ bản của giao diện

- Tạo Giao diện Người dùng (GUI): Sử dụng Tkinter để tạo một giao diện đồ họa đơn giản. Bạn tạo cửa sổ chính (root) và thêm các thành phần như các nút (Button), các nhãn (Label), và các tùy chọn (OptionMenu).

- Xử lý Sự kiện: Đặt các hàm xử lý sự kiện, như khi người dùng chọn thuật toán hoặc nhấn nút "Scramble". Các hàm này được gắn vào các sự kiện thích hợp, ví dụ: command=partial(exchange, 0).

- Sử dụng Hàm Bất đồng bộ (after): Sử dụng after để gọi hàm theo chu kỳ, giống như hàm time.sleep. Trong trường hợp này, nó được sử dụng để thực hiện các bước giải quyết một cách từ từ với khoảng thời gian nhất định giữa các bước.

- Tạo Buttons Động: Tạo các nút động trong một lưới, trong trường hợp này là các ô của trò chơi 8-Puzzle. Sử dụng Button để tạo nút và cập nhật nội dung của chúng tương ứng với trạng thái hiện tại của trò chơi.

- Lập trình Hướng đối tượng (OOP): Dùng hàm để tổ chức code, nhưng vẫn giữ một số biến và hàm toàn cục. Một số logic còn sẽ được tách thành các class và module để tạo thành cấu trúc rõ ràng hơn.

#### Định nghĩa, hiệu suất của các thuật toán

**\* Tìm kiếm theo chiều rộng - BFS:**

Tìm kiếm theo chiều rộng là một chiến lược đơn giản trong đó nút gốc được mở rộng trước, sau đó tất cả các nút kế thừa của nút gốc được mở rộng tiếp theo, sau đó là các nút kế thừa của chúng, v.v. Nói chung, tất cả các nút đều được mở rộng ở một độ sâu nhất định trong cây tìm kiếm trước khi bất kỳ nút nào ở cấp độ tiếp theo được mở rộng.

Tìm kiếm theo chiều rộng là một ví dụ của thuật toán tìm kiếm đồ thị tổng quát, trong đó nút nông nhất chưa được mở rộng được chọn để mở rộng. Điều này đạt được rất đơn giản bằng cách sử dụng hàng đợi FIFO cho đường biên. Do đó, các nút mới (luôn sâu hơn nút cha mẹ của chúng) sẽ ở phía sau hàng đợi và các nút cũ, nông hơn các nút mới, sẽ được mở rộng trước. Có một điều chỉnh nhỏ về thuật toán tìm kiếm đồ thị chung, đó là kiểm tra mục tiêu được áp dụng cho từng nút khi nó được tạo thay vì khi nó được chọn để mở rộng. Quyết định này được giải thích dưới đây, trong đó chúng tôi thảo luận về độ phức tạp về thời gian. Cũng lưu ý rằng thuật toán, tuân theo quy luật chung mẫu để tìm kiếm biểu đồ, loại bỏ mọi đường dẫn mới đến trạng thái đã có trong tập hợp biên giới hoặc đã khám phá; thật dễ dàng để thấy rằng bất kỳ con đường nào như vậy ít nhất phải sâu bằng con đường đã được tìm thấy. Do đó, tìm kiếm theo chiều rộng luôn có đường đi nông nhất tới mọi nút trên biên giới.

**\* Tìm kiếm theo chiều sâu - DFS:**

Tìm kiếm ưu tiên chiều sâu hay tìm kiếm theo chiều sâu (tiếng Anh: Depth-first search - DFS) là một thuật toán duyệt hoặc tìm kiếm trên một cây hoặc một đồ thị. Thuật toán khởi đầu tại gốc (hoặc chọn một đỉnh nào đó coi như gốc) và phát triển xa nhất có thể theo mỗi nhánh.

Thông thường, DFS là một dạng tìm kiếm thông tin không đầy đủ mà quá trình tìm kiếm được phát triển tới đỉnh con đầu tiên của nút đang tìm kiếm cho tới khi gặp được đỉnh cần tìm hoặc tới một nút không có con. Khi đó giải thuật quay lui về đỉnh vừa mới tìm kiếm ở bước trước. Trong dạng không đệ quy, tất cả các đỉnh chờ được phát triển được bổ sung vào một ngăn xếp LIFO.

**\* Giải thuật tham lam - Greedy:**

Giải thuật tham lam (tiếng Anh: Greedy algorithm) là một thuật toán giải quyết một bài toán theo kiểu metaheuristic để tìm kiếm lựa chọn tối ưu địa phương ở mỗi bước đi với hy vọng tìm được tối ưu toàn cục.

Nói chung, giải thuật tham lam có năm thành phần:

* Một tập hợp các ứng viên (candidate), để từ đó tạo ra lời giải
* Một hàm lựa chọn, để theo đó lựa chọn ứng viên tốt nhất để bổ sung vào lời giải
* Một hàm khả thi (feasibility), dùng để quyết định nếu một ứng viên có thể được dùng để xây dựng lời giải
* Một hàm mục tiêu, ấn định giá trị của lời giải hoặc một lời giải chưa hoàn chỉnh
* Một hàm đánh giá, chỉ ra khi nào ta tìm ra một lời giải hoàn chỉnh.

**\* Giải thuật Hill Climbing:**

Giải thuật Hill Climbing (giải thuật leo đồi) là một kỹ thuật tối ưu toán học theo phương pháp tìm kiếm cục bộ. Nó thực hiện tìm một trạng thái tốt hơn trạng thái hiện tại để tiến đến. Để biết trạng thái tiếp theo nào là lớn hơn, nó dùng một hàm heuristic h để xác định trạng thái nào là tốt nhất. Giải thuật Hill Climbing dễ dàng tìm thấy một giải pháp tốt cục bộ (local optimum) nhưng khó tìm thấy giải pháp tối ưu (global optinum) trong tất cả các giải pháp được đưa ra (search space). Hill Climbing phù hợp để giải các bài toán “convex” (dịch tạm: lồi) như là tìm kiếm đơn giản trong lập trình tuyến tính, tìm kiếm nhị phân.

**\* Giải thuật A\*:**

Trong khoa học máy tính, A\* (đọc là A sao) là thuật toán tìm kiếm trong đồ thị. Thuật toán này tìm một đường đi từ một nút khởi đầu tới một nút đích cho trước (hoặc tới một nút thỏa mãn một điều kiện đích). Thuật toán này sử dụng một "đánh giá heuristic" để xếp loại từng nút theo ước lượng về tuyến đường tốt nhất đi qua nút đó. Thuật toán này duyệt các nút theo thứ tự của đánh giá heuristic này. Do đó, thuật toán A\* là một ví dụ của tìm kiếm theo lựa chọn tốt nhất (best-first search).

**\* Giải thuật IDS:**

Tìm kiếm sâu lặp lại (IDS) là một chiến lược tìm kiếm biểu đồ lặp tận dụng tính hoàn chỉnh của chiến lược Tìm kiếm theo chiều rộng (BFS) nhưng sử dụng ít bộ nhớ hơn trong mỗi lần lặp (tương tự như Tìm kiếm theo chiều sâu).

IDS đạt được sự hoàn thiện mong muốn bằng cách thực thi giới hạn độ sâu trên DFS nhằm giảm thiểu khả năng bị mắc kẹt trong một nhánh vô hạn hoặc một nhánh rất dài. Nó tìm kiếm từng nhánh của một nút từ trái sang phải cho đến khi đạt đến độ sâu cần thiết. Khi đã có, IDS sẽ quay trở lại nút gốc và khám phá một nhánh khác tương tự như DFS.

**\* Giải thuật UCS:**

Tìm kiếm chi phí thống nhất (UCS) là một thuật toán tìm kiếm và truyền tải đồ thị được sử dụng trong lĩnh vực trí tuệ nhân tạo và khoa học máy tính. UCS là một thuật toán tìm kiếm có hiểu biết giúp khám phá biểu đồ bằng cách mở rộng dần các nút bắt đầu từ nút ban đầu và di chuyển về phía nút mục tiêu trong khi xem xét chi phí liên quan đến từng cạnh hoặc bước.

Thuật toán này chủ yếu được sử dụng khi chi phí bước không giống nhau nhưng chúng ta cần giải pháp tối ưu cho trạng thái mục tiêu. Trong những trường hợp như vậy, chúng tôi sử dụng Tìm kiếm chi phí thống nhất để tìm mục tiêu và đường dẫn, bao gồm chi phí tích lũy để mở rộng từng nút từ nút gốc đến nút mục tiêu. Nó không đi sâu hay rộng. Nó tìm kiếm nút tiếp theo với chi phí thấp nhất và trong trường hợp có cùng chi phí đường dẫn, hãy xem xét thứ tự từ điển trong trường hợp của chúng tôi.

#### Nền tảng kỹ thuật cơ bản của thuật toán

- Memoization (memoize): Hàm memoize được sử dụng để lưu trữ kết quả của một hàm cho các đầu vào giống nhau, giúp tăng hiệu suất bằng cách tránh việc tính toán lại những giá trị đã được tính trước đó.

- Thư viện heapq: Thư viện heapq được sử dụng để triển khai cấu trúc dữ liệu hàng đợi ưu tiên (PriorityQueue). Hàng đợi này được sử dụng trong các thuật toán tìm kiếm đồ thị.

- Lập trình hướng đối tượng: Code được tổ chức theo mô hình hướng đối tượng, với các lớp như Problem, Node, và EightPuzzle. Điều này giúp tăng cường tính tái sử dụng, tổ chức code, và hiểu quả khi giải quyết vấn đề cụ thể như trò chơi 8-Puzzle.

- Kỹ thuật Memoization trong Class (\_\_delitem\_\_): Trong lớp PriorityQueue, kỹ thuật memoization được sử dụng để xóa một phần tử khỏi hàng đợi ưu tiên.

- Lập trình Động (\_\_eq\_\_ và \_\_hash\_\_): Trong lớp Node, các phương thức \_\_eq\_\_ và \_\_hash\_\_ được triển khai để so sánh và băm các đối tượng Node. Điều này quan trọng khi chúng ta sử dụng Node trong cấu trúc dữ liệu như hàng đợi ưu tiên.

**a. Class PriorityQueue**

Class PriorityQueue này cung cấp một triển khai đơn giản và hiệu quả của hàng đợi ưu tiên sử dụng heap, phục vụ nhiều mục đích trong việc quản lý phần tử dựa trên ưu tiên.

class PriorityQueue:

    """A Queue in which the minimum (or maximum) element (as determined by f and

    order) is returned first.

    If order is 'min', the item with minimum f(x) is

    returned first; if order is 'max', then it is the item with maximum f(x).

    Also supports dict-like lookup."""

    def \_\_init\_\_(self, order='min', f=lambda x: x):

        self.heap = []

        if order == 'min':

            self.f = f

        elif order == 'max':  *# now item with max f(x)*

            self.f = lambda x: -f(x)  *# will be popped first*

        else:

            raise ValueError("Order must be either 'min' or 'max'.")

    def append(self, item):

        """Insert item at its correct position."""

        heapq.heappush(self.heap, (self.f(item), item))

    def extend(self, items):

        """Insert each item in items at its correct position."""

        for item in items:

            self.append(item)

    def pop(self):

        """Pop and return the item (with min or max f(x) value)

        depending on the order."""

        if self.heap:

            return heapq.heappop(self.heap)[1]

        else:

            raise Exception('Trying to pop from empty PriorityQueue.')

    def \_\_len\_\_(self):

        """Return current capacity of PriorityQueue."""

        return len(self.heap)

    def \_\_contains\_\_(self, key):

        """Return True if the key is in PriorityQueue."""

        return any([item == key for \_, item in self.heap])

    def \_\_getitem\_\_(self, key):

        """Returns the first value associated with key in PriorityQueue.

        Raises KeyError if key is not present."""

        for value, item in self.heap:

            if item == key:

                return value

        raise KeyError(str(key) + " is not in the priority queue")

    def \_\_delitem\_\_(self, key):

        """Delete the first occurrence of key."""

        try:

            del self.heap[[item == key for \_, item in self.heap].index(True)]

        except ValueError:

            raise KeyError(str(key) + " is not in the priority queue")

        heapq.heapify(self.heap)

**def \_\_init\_\_(self, order='min', f=lambda x: x):** Phương thức khởi tạo.

order: Xác định xem phần tử có ưu tiên nhỏ nhất hay lớn nhất sẽ được phục vụ trước. Mặc định là 'min'.

f: Hàm ưu tiên, xác định cách tính ưu tiên dựa trên giá trị của phần tử. Mặc định là hàm đồng nhất.

Trong phương thức này, một heap rỗng được tạo ra để lưu trữ các phần tử, và hàm ưu tiên (self.f) được xác định dựa trên thứ tự được chọn. Nếu là 'max', hàm ưu tiên được đảo ngược bằng cách sử dụng lambda function.

**def append(self, item):** Phương thức chèn một phần tử vào hàng đợi ưu tiên.

item: Phần tử cần chèn vào hàng đợi.

Phương thức này sử dụng heapq.heappush để chèn phần tử vào heap theo thứ tự ưu tiên.

**def extend(self, items):** Phương thức chèn một danh sách các phần tử vào hàng đợi ưu tiên.

items: Danh sách các phần tử cần chèn vào hàng đợi.

Phương thức này gọi phương thức append cho mỗi phần tử trong danh sách, đảm bảo rằng các phần tử được chèn theo thứ tự ưu tiên.

**def pop(self):** Phương thức loại bỏ và trả về phần tử có ưu tiên cao nhất hoặc thấp nhất, tùy thuộc vào thứ tự của hàng đợi.

Phương thức sử dụng heapq.heappop để loại bỏ và trả về phần tử đầu tiên của heap.

**def \_\_len\_\_(self):** Phương thức trả về số lượng phần tử hiện tại trong hàng đợi ưu tiên.

**def \_\_contains\_\_(self, key):** Phương thức kiểm tra xem một khóa có tồn tại trong hàng đợi ưu tiên hay không.

key: Khóa cần kiểm tra.

Phương thức này sử dụng một biểu thức generator và hàm any để kiểm tra xem có bất kỳ phần tử nào có giá trị bằng với khóa không.

**def \_\_getitem\_\_(self, key):** Phương thức trả về giá trị ưu tiên liên kết với một khóa trong hàng đợi ưu tiên.

key: Khóa cần lấy giá trị ưu tiên.

Phương thức này sử dụng một vòng lặp để tìm giá trị ưu tiên của phần tử đầu tiên có khóa bằng với khóa được cung cấp.

**def \_\_delitem\_\_(self, key):** Phương thức xóa sự xuất hiện đầu tiên của một khóa khỏi hàng đợi ưu tiên.

key: Khóa cần xóa.

Phương thức này sử dụng một list comprehension để tạo một danh sách các giá trị Boolean, thể hiện xem mỗi phần tử có bằng với khóa cần xóa không. Sau đó, nó sử dụng index để tìm vị trí của phần tử đầu tiên có giá trị True, và sử dụng pop để loại bỏ phần tử đó khỏi heap. Cuối cùng, nó sử dụng heapq.heapify để đảm bảo tính chất heap.

**b. Class Problem**

Lớp này được thiết kế để được kế thừa, và các phương thức của nó có thể được ghi đè để điều chỉnh cho các lĩnh vực vấn đề cụ thể. Đây là một phần của một khung công việc để giải quyết các vấn đề bằng cách sử dụng các thuật toán tìm kiếm khác nhau và cung cấp một giao diện nhất quán cho các loại vấn đề khác nhau.

class Problem:

    """The abstract class for a formal problem. You should subclass

    this and implement the methods actions and result, and possibly

    \_\_init\_\_, goal\_test, and path\_cost. Then you will create instances

    of your subclass and solve them with the various search functions."""

    def \_\_init\_\_(self, initial, goal=None):

        """The constructor specifies the initial state, and possibly a goal

        state, if there is a unique goal. Your subclass's constructor can add

        other arguments."""

        self.initial = initial

        self.goal = goal

    def actions(self, state):

        """Return the actions that can be executed in the given

        state. The result would typically be a list, but if there are

        many actions, consider yielding them one at a time in an

        iterator, rather than building them all at once."""

        possible\_actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']

        index\_blank\_square = self.find\_blank\_square(state)

        if index\_blank\_square % 3 == 0:

            possible\_actions.remove('LEFT')

        if index\_blank\_square < 3:

            possible\_actions.remove('UP')

        if index\_blank\_square % 3 == 2:

            possible\_actions.remove('RIGHT')

        if index\_blank\_square > 5:

            possible\_actions.remove('DOWN')

        return possible\_actions

    def result(self, state, action):

        """Return the state that results from executing the given

        action in the given state. The action must be one of

        self.actions(state)."""

*# blank is the index of the blank square*

        blank = self.find\_blank\_square(state)

        new\_state = list(state)

        delta = {'UP': -3, 'DOWN': 3, 'LEFT': -1, 'RIGHT': 1}

        neighbor = blank + delta[action]

        new\_state[blank], new\_state[neighbor] = new\_state[neighbor], new\_state[blank]

        return tuple(new\_state)

    def goal\_test(self, state):

        """Return True if the state is a goal. The default method compares the

        state to self.goal or checks for state in self.goal if it is a

        list, as specified in the constructor. Override this method if

        checking against a single self.goal is not enough."""

        if isinstance(self.goal, list):

            return is\_in(state, self.goal)

        else:

            return state == self.goal

    def path\_cost(self, c, state1, action, state2):

        """Return the cost of a solution path that arrives at state2 from

        state1 via action, assuming cost c to get up to state1. If the problem

        is such that the path doesn't matter, this function will only look at

        state2. If the path does matter, it will consider c and maybe state1

        and action. The default method costs 1 for every step in the path."""

        return c + 1

    def value(self, state):

        """For optimization problems, each state has a value. Hill Climbing

        and related algorithms try to maximize this value."""

        raise sum(state)

    def find\_blank\_square(self, state):

        """Return the index of the blank square in a given state"""

        return state.index(0)

**def \_\_init\_\_(self, initial, goal=None):**

Hàm khởi tạo, thiết lập vấn đề với trạng thái ban đầu và trạng thái mục tiêu tùy chọn.

Tham số:

initial: Trạng thái ban đầu của vấn đề.

goal: Trạng thái mục tiêu của vấn đề (không bắt buộc).

**def actions(self, state):**

Trả về danh sách các hành động có thể thực hiện trong trạng thái cho trước.

Triển khai mặc định cung cấp bốn hành động có thể: 'LÊN', 'XUỐNG', 'TRÁI', 'PHẢI'.

Một số hành động có thể bị hạn chế dựa trên vị trí của ô trống trong trò chơi.

**def result(self, state, action):**

Trả về trạng thái kết quả sau khi thực hiện một hành động cụ thể trong một trạng thái cho trước.

Triển khai mặc định đổi chỗ ô trống với ô láng giềng dựa trên hành động được chọn.

**def goal\_test(self, state):**

Kiểm tra xem một trạng thái có phải là trạng thái mục tiêu không.

Triển khai mặc định so sánh trạng thái với self.goal hoặc kiểm tra sự tồn tại của trạng thái trong self.goal nếu đó là một danh sách.

**def path\_cost(self, c, state1, action, state2):**

Trả về chi phí của một đường đi giải pháp từ state1 đến state2 thông qua action, giả sử chi phí c để đến state1.

Triển khai mặc định gán chi phí là 1 cho mỗi bước trên đường đi.

**def value(self, state):**

Đối với các vấn đề tối ưu hóa, trả về một giá trị liên quan đến một trạng thái.

Triển khai mặc định trả về tổng các giá trị trong trạng thái (tuy nhiên, có vẻ có một lỗi ở đây vì raise sum(state) sẽ gây lỗi).

**def find\_blank\_square(self, state):**

Trả về chỉ mục của ô trống (0) trong trạng thái cho trước.

c. Class EightPuzzle

Định nghĩa một lớp EightPuzzle (Bài toán 8 Puzzle) là một lớp con của lớp Problem. Nó mô tả bài toán trượt ô số từ 1 đến 8 trên một bảng 3x3, trong đó một ô là ô trống. Trạng thái được biểu diễn dưới dạng tuple có độ dài 9, trong đó phần tử tại chỉ mục i đại diện cho số ô tại chỉ mục i (0 nếu đó là ô trống).

class EightPuzzle(Problem):

    """ The problem of sliding tiles numbered from 1 to 8 on a 3x3 board, where one of the

    squares is a blank. A state is represented as a tuple of length 9, where  element at

    index i represents the tile number  at index i (0 if it's an empty square) """

    def \_\_init\_\_(self, initial, goal=(1, 2, 3, 4, 5, 6, 7, 8, 0)):

        """ Define goal state and initialize a problem """

        super().\_\_init\_\_(initial, goal)

    def find\_blank\_square(self, state):

        """Return the index of the blank square in a given state"""

        return state.index(0)

    def actions(self, state):

        """ Return the actions that can be executed in the given state.

        The result would be a list, since there are only four possible actions

        in any given state of the environment """

        possible\_actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']

        index\_blank\_square = self.find\_blank\_square(state)

        if index\_blank\_square % 3 == 0:

            possible\_actions.remove('LEFT')

        if index\_blank\_square < 3:

            possible\_actions.remove('UP')

        if index\_blank\_square % 3 == 2:

            possible\_actions.remove('RIGHT')

        if index\_blank\_square > 5:

            possible\_actions.remove('DOWN')

        return possible\_actions

    def result(self, state, action):

        """ Given state and action, return a new state that is the result of the action.

        Action is assumed to be a valid action in the state """

*# blank is the index of the blank square*

        blank = self.find\_blank\_square(state)

        new\_state = list(state)

        delta = {'UP': -3, 'DOWN': 3, 'LEFT': -1, 'RIGHT': 1}

        neighbor = blank + delta[action]

        new\_state[blank], new\_state[neighbor] = new\_state[neighbor], new\_state[blank]

        return tuple(new\_state)

    def goal\_test(self, state):

        """ Given a state, return True if state is a goal state or False, otherwise """

        return state == self.goal

    def check\_solvability(self, state):

        """ Checks if the given state is solvable """

        inversion = 0

        for i in range(len(state)):

            for j in range(i + 1, len(state)):

                if (state[i] > state[j]) and state[i] != 0 and state[j] != 0:

                    inversion += 1

        return inversion % 2 == 0

    def h(self, node):

        """ Return the heuristic value for a given state. Default heuristic function used is

        h(n) = number of misplaced tiles """

        return sum(s != g for (s, g) in zip(node.state, self.goal))

**def \_\_init\_\_(self, initial, goal=(1, 2, 3, 4, 5, 6, 7, 8, 0)):**

Hàm khởi tạo, định nghĩa trạng thái mục tiêu là một tuple mặc định nếu không có trạng thái mục tiêu được chỉ định.

**def find\_blank\_square(self, state):**

Trả về chỉ mục của ô trống trong một trạng thái cho trước.

**def actions(self, state):**

Trả về danh sách các hành động có thể thực hiện trong trạng thái cho trước.

Hành động bị hạn chế bởi vị trí của ô trống.

**def result(self, state, action):**

Trả về trạng thái mới sau khi thực hiện một hành động trong một trạng thái cho trước.

**def goal\_test(self, state):**

Kiểm tra xem một trạng thái có phải là trạng thái mục tiêu hay không.

**def check\_solvability(self, state):**

Kiểm tra xem một trạng thái cho trước có thể giải được hay không, sử dụng số lượt đảo để đánh giá tính giải được.

**def h(self, node):**

Hàm heuristic (hàm h) trả về giá trị heuristic cho một trạng thái dựa trên số lượng ô đặt sai vị trí so với trạng thái mục tiêu.

**\* best\_first\_graph\_search(problem, f, display=False)**

- Mục tiêu: Tìm kiếm các nút có giá trị f thấp nhất đầu tiên.

- Đầu vào:

* problem: Vấn đề tìm kiếm.
* f: Hàm giá trị đánh giá của mỗi nút.
* display: Hiển thị thông tin tìm kiếm (mặc định là False).

- Đầu ra: Trả về nút làm cho hàm đánh giá f thấp nhất.

- Mô tả: Hàm này triển khai thuật toán tìm kiếm đồ thị tốt nhất. Sử dụng hàng đợi ưu tiên để quản lý các nút, và sử dụng hàm f được đánh giá bằng memoization.

- Thuật toán:

def best\_first\_graph\_search(problem, f, display=False):

    """Search the nodes with the lowest f scores first.

    You specify the function f(node) that you want to minimize; for example,

    if f is a heuristic estimate to the goal, then we have greedy best

    first search; if f is node.depth then we have breadth-first search.

    There is a subtlety: the line "f = memoize(f, 'f')" means that the f

    values will be cached on the nodes as they are computed. So after doing

    a best first search you can examine the f values of the path returned."""

    f = memoize(f, 'f')

    node = Node(problem.initial)

    frontier = PriorityQueue('min', f)

    frontier.append(node)

    explored = set()

    while frontier:

        node = frontier.pop()

        if problem.goal\_test(node.state):

            if display:

                print(len(explored), "paths have been expanded and", len(frontier), "paths remain in the frontier")

            return node

        explored.add(node.state)

        for child in node.expand(problem):

            if child.state not in explored and child not in frontier:

                frontier.append(child)

            elif child in frontier:

                if f(child) < frontier[child]:

                    del frontier[child]

                    frontier.append(child)

    return None

**\* uniform\_cost\_search(problem, display=False)**

- Mục tiêu: Tìm kiếm đồ thị với chi phí đường đi thấp nhất.

- Đầu vào:

* problem: Vấn đề tìm kiếm.
* display: Hiển thị thông tin tìm kiếm (mặc định là False).

- Đầu ra: Trả về nút làm cho chi phí đường đi thấp nhất.

- Mô tả: Sử dụng best\_first\_graph\_search với hàm đánh giá là chi phí đường đi của mỗi nút.

- Thuật toán:

#UCS

def uniform\_cost\_search(problem, display=False):

    """[Figure 3.14]"""

    return best\_first\_graph\_search(problem, lambda node: node.path\_cost, display)

Hàm uniform\_cost\_search được thiết kế để thực hiện tìm kiếm chi phí đồng đều (uniform cost search), một loại tìm kiếm đầu tiên tốt nhất (best-first search) mà ưu tiên các đường đi có chi phí thấp nhất.

Hàm uniform\_cost\_search được định nghĩa với một tham số bắt buộc (problem) và một tham số tùy chọn (display mặc định là False).

best\_first\_graph\_search: Gọi hàm best\_first\_graph\_search với ba đối số - problem, hàm ước lượng chi phí được định nghĩa bởi lambda node: node.path\_cost, và display.

lambda node: node.path\_cost: Hàm ẩn danh (lambda) nhận một đối tượng node và trả về giá trị của thuộc tính path\_cost của nó. Trong trường hợp này, nó sử dụng chi phí của đường đi từ trạng thái ban đầu đến nút hiện tại làm giá trị ước lượng chi phí.

display: Tham số tùy chọn, mặc định là False. Nếu được đặt là True, các thông báo về số lượng đường đã mở rộng và số lượng đường còn lại trong frontier sẽ được hiển thị.

return: Kết quả của uniform\_cost\_search là kết quả của best\_first\_graph\_search. Nếu tìm kiếm tìm thấy trạng thái mục tiêu, nó sẽ trả về một đối tượng Node đại diện cho trạng thái mục tiêu. Nếu không tìm thấy, hàm sẽ trả về None.

**\* astar\_search(problem, h=None, display=False)**

- Mục tiêu: Tìm kiếm đồ thị với chi phí đường đi và hàm heuristic.

- Đầu vào:

* problem: Vấn đề tìm kiếm.
* h: Hàm heuristic (nếu không có, sử dụng problem.h).
* display: Hiển thị thông tin tìm kiếm (mặc định là False).

- Đầu ra: Trả về nút làm cho giá trị f(n) = g(n) + h(n) thấp nhất.

- Mô tả: Sử dụng best\_first\_graph\_search với hàm đánh giá là tổng chi phí đường đi và hàm heuristic.

- Thuật toán:

# A\*

def astar\_search(problem, h=None, display=False):

    """A\* search is best-first graph search with f(n) = g(n) + h(n).

    You need to specify the h function when you call astar\_search, or

    else in your Problem subclass."""

    h = memoize(h or problem.h, 'h')

return best\_first\_graph\_search(problem, lambda n: n.path\_cost + h(n), display)

Hàm astar\_search được định nghĩa với ba tham số - problem, h là hàm ước lượng chi phí đến mục tiêu (mặc định là None), và display là tham số tùy chọn để hiển thị thông tin (mặc định là False).

Dòng h = memoize(h or problem.h, 'h') thiết lập hàm ước lượng chi phí h sử dụng memoize. Nếu h là None, nó sẽ sử dụng hàm ước lượng chi phí mặc định từ problem.h. Hàm memoize được sử dụng để lưu trữ kết quả của các cuộc gọi trước đó Dòng gọi hàm best\_first\_graph\_search với ba đối số - problem, hàm ước lượng chi phí được định nghĩa bởi lambda n: n.path\_cost + h(n), và display. Hàm ước lượng chi phí này là f(n)=g(n)+h(n), nơi g(n) là chi phí thực tế để đến nút n, h(n) là hàm ước lượng chi phí từ nút n đến mục tiêu, và f(n) là tổng chi phí ước lượng. Kết quả của hàm best\_first\_graph\_search được trả về.để tối ưu hóa hiệu suất.

Top of Form

**\* greedy\_best\_first\_graph\_search(problem, h=None, display=False)**

- Mục tiêu: Tìm kiếm đồ thị với giá trị heuristic thấp nhất.

- Đầu vào:

* problem: Vấn đề tìm kiếm.
* h: Hàm heuristic (nếu không có, sử dụng problem.h).
* display: Hiển thị thông tin tìm kiếm (mặc định là False).

- Đầu ra: Trả về nút làm cho giá trị h(n) thấp nhất.

- Mô tả: Sử dụng best\_first\_graph\_search với hàm đánh giá là giá trị heuristic.

- Thuật toán:

# Greedy

def greedy\_best\_first\_graph\_search(problem, h=None, display=False):

    """Greedy search is best-first graph search with f(n) = h(n).

    You need to specify the h function when you call greedy, or

    else in your Problem subclass."""

    h = memoize(h or problem.h, 'h')

    return best\_first\_graph\_search(problem, lambda n: h(n), display)

Hàm greedy\_best\_first\_graph\_search thực hiện thuật toán tìm kiếm Greedy, một thuật toán tìm kiếm đầu tiên tốt nhất sử dụng chỉ hàm ước lượng chi phí h(n), nơi h(n) là ước lượng chi phí từ nút n đến mục tiêu.

Hàm greedy\_best\_first\_graph\_search được định nghĩa với ba tham số - problem, h là hàm ước lượng chi phí đến mục tiêu (mặc định là None), và display là tham số tùy chọn để hiển thị thông tin (mặc định là False).

Dòng **h = memoize(h or problem.h, 'h')** thiết lập hàm ước lượng chi phí h sử dụng memoize. Nếu h là None, nó sẽ sử dụng hàm ước lượng chi phí mặc định từ problem.h. Hàm memoize được sử dụng để lưu trữ kết quả của các cuộc gọi trước đó để tối ưu hóa hiệu suất.

Dòng **return best\_first\_graph\_search(problem, lambda n: h(n), display)** gọi hàm best\_first\_graph\_search với ba đối số - problem, hàm ước lượng chi phí được định nghĩa bởi lambda n: h(n), và display. Hàm ước lượng chi phí này là (n)=h(n), nơi h(n) là ước lượng chi phí từ nút n đến mục tiêu. Kết quả của hàm best\_first\_graph\_search được trả về.

**\* depth\_first\_graph\_search(problem)**

- Mục tiêu: Tìm kiếm các nút ở độ sâu lớn nhất đầu tiên.

- Đầu vào:

problem: Vấn đề tìm kiếm.

- Đầu ra: Trả về nút làm cho độ sâu lớn nhất.

- Mô tả: Sử dụng thuật toán tìm kiếm theo độ sâu (DFS) để tìm kiếm đồ thị.

- Thuật toán:

#DFS

def depth\_first\_graph\_search(problem):

    """

    [Figure 3.7]

    Search the deepest nodes in the search tree first.

    Search through the successors of a problem to find a goal.

    The argument frontier should be an empty queue.

    Does not get trapped by loops.

    If two paths reach a state, only use the first one.

    """

    frontier = [(Node(problem.initial))]  # Stack

    explored = set()

    while frontier:

        node = frontier.pop()

        if problem.goal\_test(node.state):

            return node

        explored.add(node.state)

        frontier.extend(child for child in node.expand(problem)

                        if child.state not in explored and child not in frontier)

    return None

Khởi Tạo Frontier: frontier là một danh sách (stack) chứa các nút của cây tìm kiếm. Nút ban đầu được thêm vào danh sách.

Khởi Tạo Explored: explored là một tập hợp lưu trữ các trạng thái đã được kiểm tra để tránh lặp lại.

Vòng Lặp Chính (While Frontier): Vòng lặp chạy cho đến khi frontier trở thành rỗng.

node = frontier.pop(): Lấy nút cuối cùng từ frontier (cấu trúc stack).

if problem.goal\_test(node.state): return node: Kiểm tra xem trạng thái của nút hiện tại có phải là trạng thái mục tiêu không. Nếu có, trả về nút này.

explored.add(node.state): Thêm trạng thái của nút vào explored để tránh lặp lại.

frontier.extend(...): Mở rộng frontier bằng cách thêm các nút con của nút hiện tại vào danh sách nếu chúng chưa được kiểm tra và không nằm trong frontier.

Trả về None: Nếu frontier trở thành rỗng mà không tìm thấy trạng thái mục tiêu, hàm trả về None.

**\* breadth\_first\_graph\_search(problem):**

- Mục tiêu: Tìm kiếm các nút ở độ sâu thấp nhất đầu tiên.

- Đầu vào:

problem: Vấn đề tìm kiếm.

- Đầu ra: Trả về nút làm cho độ sâu thấp nhất.

- Mô tả: Sử dụng thuật toán tìm kiếm theo bề rộng (BFS) để tìm kiếm đồ thị.

- Thuật toán:

#BFS

def breadth\_first\_graph\_search(problem):

    """[Figure 3.11]

    Note that this function can be implemented in a

    single line as below:

    return graph\_search(problem, FIFOQueue())

    """

    node = Node(problem.initial)

    if problem.goal\_test(node.state):

        return node

    frontier = deque([node])

    explored = set()

    while frontier:

        node = frontier.popleft()

        explored.add(node.state)

        for child in node.expand(problem):

            if child.state not in explored and child not in frontier:

                if problem.goal\_test(child.state):

                    return child

                frontier.append(child)

    return None

Khởi tạo frontier: Frontier được biểu diễn bằng một danh sách (deque) chứa các nút của cây tìm kiếm. Ban đầu, nút đầu tiên được thêm vào danh sách.

Khởi tạo explored: Explored là một tập hợp lưu trữ các trạng thái đã được kiểm tra để tránh lặp lại.

Vòng lặp chính (while frontier): Vòng lặp chạy cho đến khi frontier trở thành rỗng. Trong mỗi lần lặp, một nút được lấy ra từ frontier để kiểm tra.

Lấy nút từ frontier (node = frontier.pop()): Lấy nút cuối cùng từ frontier, sử dụng cấu trúc dữ liệu deque như một stack.

Kiểm tra trạng thái mục tiêu (if problem.goal\_test(node.state): return node): Kiểm tra xem trạng thái của nút hiện tại có phải là trạng thái mục tiêu không. Nếu có, hàm trả về ngay lập tức với nút hiện tại.

Thêm trạng thái vào explored (explored.add(node.state)): Thêm trạng thái của nút vào explored để đảm bảo không lặp lại việc kiểm tra cho cùng một trạng thái.

Mở rộng frontier (frontier.extend(...)): Mở rộng frontier bằng cách thêm các nút con của nút hiện tại vào danh sách, với điều kiện là chúng chưa được kiểm tra và không nằm trong frontier. Điều này đảm bảo rằng các trạng thái mới được thêm vào frontier để duyệt tiếp.

Trả về none (trong trường hợp không tìm thấy): Nếu sau khi duyệt hết tất cả các nút mà vẫn không tìm thấy trạng thái mục tiêu, hàm trả về None để thể hiện rằng không có giải pháp nào được tìm thấy.

**\* recursive\_DLS(node, problem, limit)**

- Mục tiêu: Tìm kiếm độ sâu giới hạn đệ quy.

- Đầu vào:

* node: Nút bắt đầu.
* problem: Vấn đề tìm kiếm.
* limit: Giới hạn độ sâu.

- Đầu ra: Trả về nút mục tiêu hoặc 'cutoff' nếu chạm giới hạn.

- Mô tả: Thực hiện tìm kiếm độ sâu giới hạn đệ quy.

- Thuật toán:

# IDS

def recursive\_DLS(node, problem, limit):

    if problem.goal\_test(node.state):   # Nếu vị trí đang xét là goal thì trả về node chứa kết quả

        return node

    elif limit == 0:    # Là cột mốc đánh dấu vị trí có thể là gốc hoặc goal hoặc vị trí giới hạn của độ sâu có thể đạt tới

        return 'cutoff'

    else:

        cutoff\_occurred = False

        for child in node.expand(problem):  # Mở rộng ra các node con là hậu duệ của vị trí hiện tại

            result = recursive\_DLS(child, problem, limit - 1)   # đệ quy với từng hậu duệ của vị trí hiện tại để tiếp tục tìm ra các hậu duệ tiếp theo rồi tìm ra được con đường đúng

            if result == 'cutoff': # Kiểm tra xem đã chạm tới giới hạn của độ sâu chưa

                cutoff\_occurred = True

            elif result != []:  # Kiểm tra xem đã chạm đến đích chưa

                return result

        if cutoff\_occurred:

            return 'cutoff' # Trả về đã chạm tới giới hạn

        else:

            return []   # Trả về chưa tìm được đường

Hàm recursive\_DLS:

Input:

* node: Đối tượng Node đại diện cho trạng thái hiện tại trong không gian trạng thái.
* problem: Đối tượng biểu diễn vấn đề tìm kiếm, có thể chứa các phương thức như goal\_test và expand.
* limit: Giới hạn độ sâu tối đa mà thuật toán sẽ duyệt xuống trong cây tìm kiếm.

Output:

* Nếu trạng thái hiện tại là trạng thái đích (goal), hàm trả về node.
* Nếu đã đạt đến giới hạn độ sâu (limit == 0), hàm trả về giá trị 'cutoff' để chỉ ra rằng đã chạm đến giới hạn độ sâu.
* Nếu không phải trạng thái đích và chưa đạt đến giới hạn độ sâu, hàm sẽ tiếp tục triển khai DLS đệ quy.

Quá trình DLS đệ quy:

* Nếu trạng thái hiện tại là trạng thái đích, hàm trả về node chứa kết quả.
* Nếu đạt đến giới hạn độ sâu (limit == 0), hàm trả về 'cutoff'.
* Nếu không phải trạng thái đích và chưa đạt đến giới hạn độ sâu, hàm mở rộng trạng thái hiện tại để tạo ra các trạng thái con (hậu duệ).
* Đối với mỗi trạng thái con, hàm gọi đệ quy với giới hạn giảm đi 1 (limit - 1).
* Nếu kết quả từ đệ quy là 'cutoff', đặt cutoff\_occurred = True.
* Nếu kết quả từ đệ quy không rỗng (không phải 'cutoff'), trả về kết quả đó.
* Nếu không có 'cutoff' và không có kết quả từ trạng thái con, hàm trả về một danh sách rỗng.

Trạng thái cutoff\_occurred: biến cutoff\_occurred được sử dụng để theo dõi xem có bất kỳ trạng thái con nào đã chạm đến giới hạn độ sâu chưa.

Kết quả của hàm:

* Nếu có bất kỳ 'cutoff' nào xuất hiện trong các trạng thái con, hàm trả về 'cutoff'.
* Nếu có bất kỳ kết quả nào từ trạng thái con không phải 'cutoff', hàm trả về kết quả đó.
* Nếu không có 'cutoff' và không có kết quả từ trạng thái con, hàm trả về danh sách rỗng.

**\* depth\_limited\_search(problem, limit=50)**

- Mục tiêu: Tìm kiếm độ sâu giới hạn.

- Đầu vào:

* problem: Vấn đề tìm kiếm.
* limit: Giới hạn độ sâu.

- Đầu ra: Trả về nút mục tiêu hoặc 'cutoff' nếu chạm giới hạn.

- Mô tả: Thực hiện tìm kiếm độ sâu giới hạn.

- Thuật toán:

''' IMPLEMENT THE FOLLOWING FUNCTION '''

def depth\_limited\_search(problem, limit=50):

    """See [Figure 3.17] for the algorithm"""

    return recursive\_DLS(Node(problem.initial), problem, limit) # Tùy vào độ sâu ban đầu mà thiết lập hàm đệ quy

Phân tích chi tiết code.

Input:

* problem: Đối tượng biểu diễn vấn đề tìm kiếm, chứa các phương thức như initial (trạng thái ban đầu) và goal\_test.
* limit: Giới hạn độ sâu tối đa mà thuật toán sẽ duyệt xuống trong cây tìm kiếm. Giá trị mặc định là 50.

Output:

* Kết quả của hàm recursive\_DLS, là một đối tượng Node chứa trạng thái đích nếu tìm thấy, hoặc giá trị 'cutoff' nếu đạt đến giới hạn độ sâu.

Hàm này đơn giản chỉ là một giao diện để gọi hàm đệ quy recursive\_DLS với đối tượng Node ban đầu và giới hạn độ sâu được cung cấp.

**\* iterative\_deepening\_search(problem)**

- Mục tiêu: Tìm kiếm đồ thị với độ sâu tăng dần.

- Đầu vào:

* problem: Vấn đề tìm kiếm.

- Đầu ra: Trả về nút mục tiêu hoặc 'cutoff'.

-Mô tả: Thực hiện tìm kiếm đồ thị với độ sâu tăng dần.

- Thuật toán:

''' IMPLEMENT THE FOLLOWING FUNCTION '''

def iterative\_deepening\_search(problem):

    """See [Figure 3.18] for the algorithm"""

    for depth in range(0, sys.maxsize):  #Thiết lập vòng lặp tăng độ sâu lên một để truy tìm được đường đi tối ưu nhất có thể nếu chưa tìm được đường đi tối ưu ở độ sâu hiện tại

        result = depth\_limited\_search(problem, depth) # Gọi đến hàm xử lí việc tìm kiếm có giới hạn độ sâu

        if result != 'cutoff':  # Trả ra kết quả

            return result

Hàm iterative\_deepening\_search triển khai thuật toán tìm kiếm theo độ sâu có giới hạn với chiến lược tăng dần độ sâu cho đến khi tìm thấy trạng thái đích hoặc đạt đến giới hạn độ sâu tối đa. Sau đây là phân tích chi tiết đoạn code:

Input: problem đối tượng biểu diễn vấn đề tìm kiếm, chứa các phương thức như initial (trạng thái ban đầu) và goal\_test.

Output: kết quả của hàm depth\_limited\_search, là một đối tượng Node chứa trạng thái đích nếu tìm thấy, hoặc giá trị 'cutoff' nếu đạt đến giới hạn độ sâu.

Cách hoạt động:

* Sử dụng vòng lặp for để thực hiện tìm kiếm theo độ sâu có giới hạn với chiến lược tăng dần độ sâu.
* Gọi hàm depth\_limited\_search với vấn đề và giới hạn độ sâu hiện tại.
* Nếu kết quả khác 'cutoff', nghĩa là đã tìm thấy trạng thái đích, trả về kết quả đó.
* Nếu kết quả là 'cutoff', tiếp tục với độ sâu tăng lên 1 và thực hiện lại vòng lặp.

Hàm này sử dụng chiến lược tăng dần độ sâu để duyệt qua các độ sâu từ 0 đến vô cùng, mục tiêu là tìm kiếm đường đi tối ưu nhất có thể.

**\* hill\_climbing(problem)**

- Mục tiêu: Tìm kiếm đồ thị sử dụng thuật toán Hill Climbing.

- Đầu vào:

* problem: Vấn đề tìm kiếm.

- Đầu ra: Trả về nút mục tiêu.

- Mô tả: Sử dụng thuật toán Hill Climbing để tìm kiếm đường đi tốt nhất.

- Thuật toán:

# Hill climbing

def hill\_climbing(problem):

    current = Node(problem.initial)

    current\_values = manhattan\_distance(current.state, problem.goal)

best\_neighbor = choose\_best\_neighbor(problem, current)

    while manhattan\_distance(best\_neighbor.state, problem.goal) < current\_values:

        if problem.goal\_test(best\_neighbor.state):

            return best\_neighbor

        current = best\_neighbor

        current\_values = manhattan\_distance(best\_neighbor.state, problem.goal)

        best\_neighbor = choose\_best\_neighbor(problem, current)

    return best\_neighbor

Input: problem đối tượng biểu diễn vấn đề tìm kiếm, chứa các phương thức như initial (trạng thái ban đầu), goal\_test, và choose\_best\_neighbor.

Output: đối tượng Node chứa trạng thái đích nếu tìm thấy, hoặc trạng thái cuối cùng nếu không thể di chuyển đến một trạng thái hàng xóm tốt hơn.

Cách hoạt động:

* Bắt đầu từ trạng thái ban đầu và tính giá trị đánh giá của nó (current\_values).
* Chọn trạng thái hàng xóm tốt nhất (best\_neighbor) bằng cách sử dụng hàm choose\_best\_neighbor.
* Trong khi giá trị của trạng thái hàng xóm tốt hơn (manhattan\_distance(best\_neighbor.state, problem.goal)) nhỏ hơn giá trị của trạng thái hiện tại (current\_values):
  + Nếu trạng thái hàng xóm là trạng thái đích, trả về nó.
  + Cập nhật current thành trạng thái hàng xóm và current\_values thành giá trị của trạng thái hàng xóm.
  + Chọn lại trạng thái hàng xóm tốt nhất.
  + Trả về trạng thái cuối cùng nếu không tìm thấy trạng thái đích.

**\* choose\_best\_neighbor(problem, current)**

- Mục tiêu: Chọn hàng xóm tốt nhất cho thuật toán Hill Climbing.

- Đầu vào:

problem: Vấn đề tìm kiếm.

current: Nút hiện tại.

- Đầu ra: Trả về nút hàng xóm tốt nhất.

- Mô tả: Chọn hàng xóm có giá trị heuristic thấp nhất.

- Thuật toán:

def choose\_best\_neighbor(problem, current):

    neighbors = current.expand(problem)

    best\_neighbor = neighbors[0]

    for i in range(0, len(neighbors), 1):

        if manhattan\_distance(neighbors[i].state, problem.goal) < manhattan\_distance(best\_neighbor.state, problem.goal):

            best\_neighbor = neighbors[i]

    return best\_neighbor

Chi tiết đoạn code.

Input:

* problem: Đối tượng biểu diễn vấn đề tìm kiếm, có thể chứa các phương thức như goal (trạng thái mục tiêu) và expand (phương thức mở rộng trạng thái hiện tại).
* current: Đối tượng Node đại diện cho trạng thái hiện tại.

Output: trạng thái hàng xóm tốt nhất, dựa trên khoảng cách Manhattan đến trạng thái mục tiêu.

Cách hoạt động:

* Lấy danh sách các trạng thái hàng xóm bằng cách gọi phương thức expand trên current.
* Khởi tạo best\_neighbor là trạng thái hàng xóm đầu tiên.
* Duyệt qua danh sách các trạng thái hàng xóm:
* So sánh khoảng cách Manhattan từ trạng thái hiện tại đến mục tiêu với khoảng cách của best\_neighbor đến mục tiêu.
* Nếu khoảng cách của trạng thái hiện tại nhỏ hơn, cập nhật best\_neighbor thành trạng thái hiện tại.
* Trả về best\_neighbor là trạng thái hàng xóm tốt nhất.

**\* manhattan\_distance(state, goal\_state)**

- Mục tiêu: Tính toán giá trị heuristic là khoảng cách Manhattan giữa trạng thái hiện tại và trạng thái mục tiêu.

- Đầu vào:

* state: Trạng thái hiện tại.
* goal\_state: Trạng thái mục tiêu.

- Đầu ra: Trả về giá trị heuristic.

- Mô tả: Tính toán khoảng cách Manhattan giữa hai trạng thái.

- Thuật toán:

def manhattan\_distance(state, goal\_state):

    # Calculate the Manhattan distance heuristic

    total\_distance = 0

    for i in range(len(state)):

        if state[i] == 0:

            continue

        for j in range(len(state)):

            if state[i] == goal\_state[j]:

                total\_distance += abs(j - i)

    return total\_distance

Phân tích chi tiết đoạn code.

Input:

* state: Một trạng thái hiện tại, được biểu diễn dưới dạng một dãy giá trị.
* goal\_state: Một trạng thái mục tiêu, cũng được biểu diễn dưới dạng một dãy giá trị.

Output: khoảng cách Manhattan giữa state và goal\_state.

Cách hoạt động:

* Duyệt qua từng phần tử của state:
* Nếu giá trị là 0 (đại diện cho ô trống), bỏ qua.
* Duyệt qua từng phần tử của goal\_state để tìm vị trí của giá trị trong state.
* Tính khoảng cách Manhattan bằng cách lấy giá trị tuyệt đối của hiệu của vị trí trong state và goal\_state, và cộng vào total\_distance.
* Trả về tổng khoảng cách Manhattan.

#### Hệ thống thư việc sử dụng

- Trong file chính:

# Nạp folder chứa 8puzzleGui

import EightPuzzleGui

- Trong thư mục hỗ trợ (EightPuzzleGui):

import functools

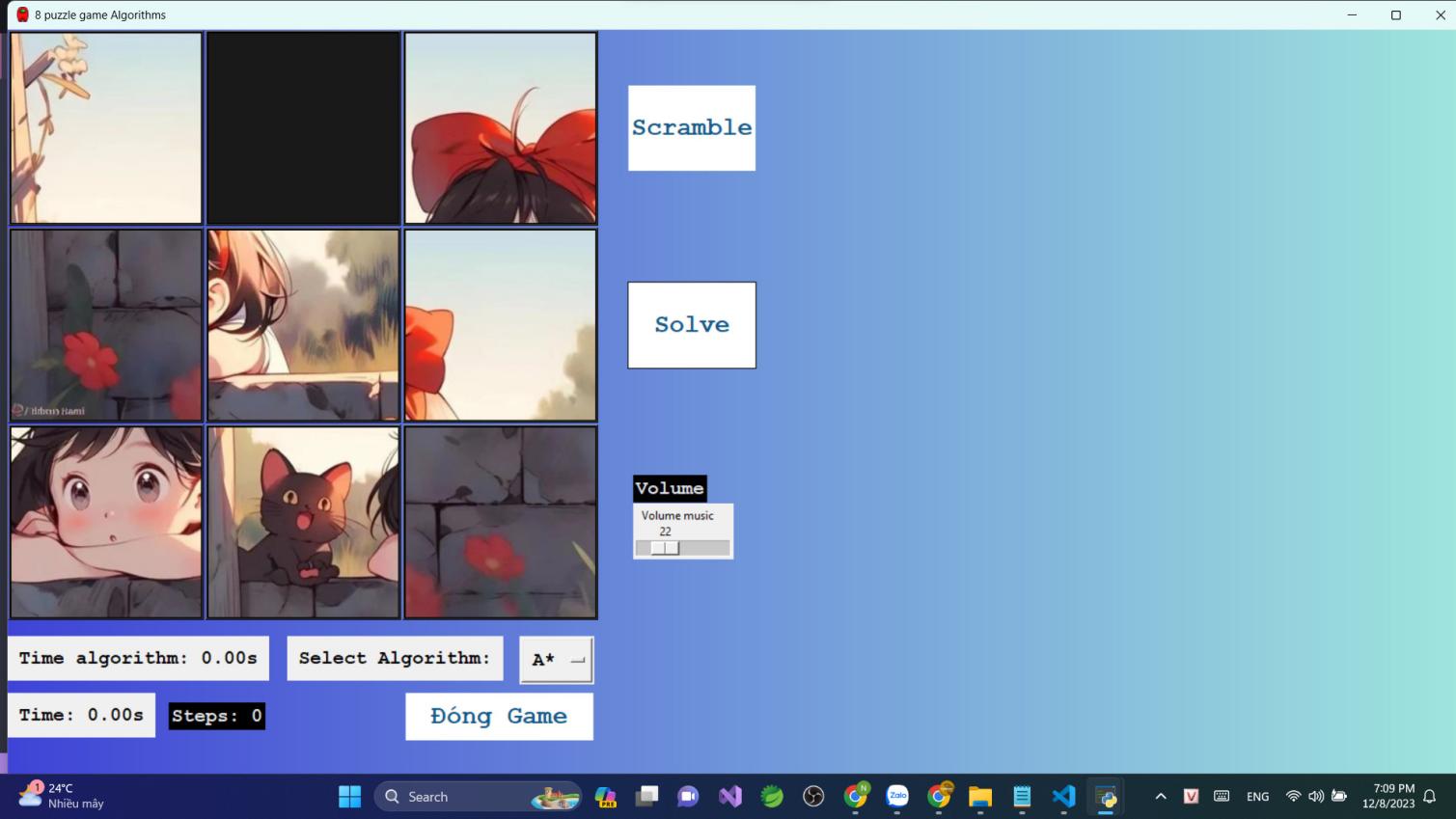
import sys

from collections import deque

import random

import heapq

#### Thiết kế giao diện.



- Nút Scramble: Dùng để xáo trộn, đảo lộn thứ tự của các mảnh ghép trong game 8 - puzzle

- Steps: tương trưng cho các bước mà khi thuật toán chạy sẽ tiêu tốn tài nguyên

- Solve: là nút dùng để bắt đầu giải ma trận 8 - puzzle khi đã chọn thuật toán

- Select Algorithm: dùng để chỉ cho nút kế bên rằng thuật toán bạn đang chọn là, ấn vào nút kế bên để chọn

#### Mã lệnh lập trình(python) - Full code

1. *#%%*
2. from Algorithm import \*
3. import os.path
4. import random
5. import time
6. from functools import partial
7. from tkinter import \*
8. import numpy as np
9. from PIL import Image, ImageTk
10. sys.path.append(os.path.join(os.path.dirname(\_\_file\_\_), '..'))
11. *#*
12. bgbtn = "#000"
13. ctxt = "#f6993f"
14. root = Tk()
15. *# Lấy kích thước của màn hình*
16. screen\_width = root.winfo\_screenwidth()
17. screen\_height = root.winfo\_screenheight()
18. root.geometry("{0}x{1}+0+0".format(screen\_width,screen\_height))
19. *# Tạo hình ảnh gradient*
20. def create\_gradient\_image(width, height, color1, color2):
21. img = Image.new('RGB', (width, height), color1)
22. img\_data = np.array(img)
23. for i in range(height):
24. img\_data[i, :, 0] = np.linspace(color1[0], color2[0], width)
25. img\_data[i, :, 1] = np.linspace(color1[1], color2[1], width)
26. img\_data[i, :, 2] = np.linspace(color1[2], color2[2], width)
27. gradient\_img = Image.fromarray(np.uint8(img\_data))
28. return gradient\_img
29. *# Màu gradient*
30. *# color\_start =  (255,255,255)*
31. *# color\_end =  (255,255,255)*
32. color\_start =  (63,68,214)
33. color\_end =  (160,230,222)
34. ffm = 'Courier New'
35. *# Tạo hình ảnh gradient*
36. gradient\_image = create\_gradient\_image(root.winfo\_screenwidth() , root.winfo\_screenheight(), color\_start, color\_end)
37. tk\_gradient\_image = ImageTk.PhotoImage(gradient\_image)
38. *# Chèn hình ảnh gradient vào cửa sổ*
39. background\_label = Label(root, image=tk\_gradient\_image)
40. background\_label.place(x=0, y=0, relwidth=1, relheight=1)
41. *# Chọn màu nền cho cửa sổ*
42. background\_color = "#fff"  *# Thay đổi mã màu theo ý muốn của bạn*
43. *# Đặt màu nền cho cửa sổ*
44. root.configure(bg=background\_color)
45. root.title("8 puzzle game Algorithms")
46. state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
47. puzzle = EightPuzzle(tuple(state))
48. solution = None
49. b = [None] \* 9
50. size = 100
51. distanceBtn = 1
52. *# Số bước di chuyển*
53. moves = 0
54. board = []
55. *# Thêm danh sách các thuật toán*
56. algorithms = ['A\*', 'BFS', 'Greedy', 'HillClimbing', 'IDS', 'DFS', 'UCS']  *# Thêm các thuật toán mà bạn muốn hỗ trợ*
57. *# Biến để theo dõi thuật toán được chọn*
58. selected\_algorithm = 'A\*'  *# Thiết lập một giá trị mặc định*
59. *# TODO: refactor into OOP, remove global variables*
60. *# Create a label to display the step count*
61. step\_label = Label(root, text="Steps: 0", font=(ffm, 15, 'bold'),fg="#E4E6EB",bg="#000")
62. *# step\_label.grid(row=0, column=7, pady=10)*
63. step\_label.place(x=170, y=710)
64. *# Tạo nút hoặc tùy chọn để chọn thuật toán*
65. algorithm\_label = Label(root, text="Select Algorithm:", font=(ffm, 13, 'bold'))
66. *# algorithm\_label.grid(row=2, column=5, pady=10)*
67. algorithm\_label.place(x=295, y=640)
68. *# Create a variable to store the selected algorithm*
69. selected\_algorithm\_var = StringVar(root)
70. selected\_algorithm\_var.set(selected\_algorithm)  *# Set the default algorithm*
71. algorithm\_option = OptionMenu(root, selected\_algorithm\_var, \*algorithms)
72. algorithm\_option.config(font=(ffm, 15, 'bold'))
73. *# algorithm\_option.grid(row=2, column=7, pady=10)*
74. algorithm\_option.place(x=495, y=633)
75. *# Tạo một biến toàn cục để theo dõi thời gian*
76. elapsed\_time\_var = StringVar()
77. elapsed\_time\_var.set("Time: 0.00s")
78. *# Tạo một label để hiển thị thời gian*
79. time\_label = Label(root, textvariable=elapsed\_time\_var, font=(ffm, 15, 'bold'),padx=10,pady=10)
80. *# time\_label.grid(row=1, column=7, pady=10)*
81. time\_label.place(x=0, y=700)
82. elapsed\_time\_algorithm\_var = StringVar()
83. elapsed\_time\_algorithm\_var.set("Time algorithm: 0.00s")
84. *# Tạo một label để hiển thị thời gian thuật toán*
85. time\_label\_algorithm = Label(root, textvariable=elapsed\_time\_algorithm\_var, font=(ffm, 15, 'bold'),padx=10,pady=10)
86. *# time\_label.grid(row=1, column=7, pady=10)*
87. time\_label\_algorithm.place(x=0, y=640)
88. def update\_selected\_algorithm(\*args):
89. global selected\_algorithm
90. selected\_algorithm = selected\_algorithm\_var.get()
91. *# Attach the function to update the selected algorithm when the user makes a selection*
92. selected\_algorithm\_var.trace("w", update\_selected\_algorithm)
93. *# Gắn hàm xử lý sự kiện chọn thuật toán cho nút tùy chọn*
94. algorithm\_option.bind("<Configure>", lambda event: select\_algorithm(selected\_algorithm\_var.get()))
95. def update\_step\_label():
96. step\_label.config(text=f"Steps: {moves}")
97. *# Hàm xử lý sự kiện khi người dùng chọn thuật toán*
98. def select\_algorithm(algorithm):
99. global selected\_algorithm
100. selected\_algorithm = algorithm
101. *# Cập nhật giao diện người dùng (nếu cần)*
102. image\_paths = [
103. "./imageonline/empty.jpg",
104. "./imageonline/01.jpg",
105. "./imageonline/02.jpg",
106. "./imageonline/03.jpg",
107. "./imageonline/04.jpg",
108. "./imageonline/05.jpg",
109. "./imageonline/06.jpg",
110. "./imageonline/07.jpg",
111. "./imageonline/08.jpg",
112. *# "./imageonline/09.jpg",*
113. *# ... Thêm đường dẫn cho các hình ảnh còn lại*
114. ]
115. *# Tạo danh sách để lưu trữ các đối tượng PhotoImage*
116. tk\_images = []
117. *# Tạo và chèn ảnh vào nút trong vòng lặp*
118. for i, image\_path in enumerate(image\_paths):
119. *# Đọc ảnh từ đường dẫn*
120. image = Image.open(image\_path)
121. resized\_image = image.resize((size, size))  *# Chỉnh kích thước ảnh*
122. tk\_image = ImageTk.PhotoImage(resized\_image)
123. *# Chuyển đổi ảnh thành đối tượng PhotoImage*
124. *# Thêm tk\_image vào danh sách*
125. tk\_images.append(tk\_image)
126. def scramble():
127. """Scrambles the puzzle starting from the goal state"""
128. global state
129. global puzzle
130. global moves  *# Declare moves as a global variable*
131. possible\_actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']
132. scramble = []
133. for \_ in range(60):
134. scramble.append(random.choice(possible\_actions))
135. for move in scramble:
136. if move in puzzle.actions(state):
137. state = list(puzzle.result(state, move))
138. puzzle = EightPuzzle(tuple(state))
139. create\_buttons()
140. moves = 0
141. update\_step\_label()
142. *# Đặt lại thời gian sau mỗi bước*
143. start\_time = time.time()
144. def solve():
145. """Solves the puzzle using the selected algorithm"""
146. if selected\_algorithm == 'A\*':
147. *# Sử dụng thuật toán A\**
148. return astar\_search(puzzle).solution()
149. elif selected\_algorithm == 'BFS':
150. *# Sử dụng thuật toán BFS*
151. return breadth\_first\_graph\_search(puzzle).solution()
152. elif selected\_algorithm == 'Greedy':
153. *# Sử dụng thuật toán Greedy*
154. return greedy\_best\_first\_graph\_search(puzzle).solution()
155. elif selected\_algorithm == 'HillClimbing':
156. *# Sử dụng thuật toán Hill Climbing*
157. return hill\_climbing(puzzle).solution()
158. elif selected\_algorithm == 'IDS':
159. return iterative\_deepening\_search(puzzle).solution()
160. elif selected\_algorithm == 'DFS':
161. return depth\_first\_graph\_search(puzzle).solution()
162. elif selected\_algorithm == 'UCS':
163. return uniform\_cost\_search(puzzle).solution()
164. def solve\_steps():
165. """Solves the puzzle step by step"""
166. global puzzle
167. global solution
168. global state
169. global moves
170. *# Bắt đầu đo thời gian*
171. start\_time = time.time()
172. solution = solve()
173. print(solution)
174. elapsed\_time\_algorithm\_var.set(f"Time algorithm: {(time.time() -start\_time):.2f}s")
175. for move in solution:
176. state = puzzle.result(state, move)
177. create\_buttons()
178. moves += 1
179. update\_step\_label()
180. root.update()
181. *# Lấy thời gian hiện tại*
182. current\_time = time.time()
183. *# Tính thời gian đã trôi qua từ khi bắt đầu đo thời gian*
184. elapsed\_time = current\_time - start\_time
185. *# Cập nhật giá trị thời gian cho label*
186. elapsed\_time\_var.set(f"Time: {elapsed\_time:.2f}s")
187. root.after(1, time.sleep(1))
188. def exchange(index):
189. """Interchanges the position of the selected tile with the zero tile under certain conditions"""
190. global state
191. global solution
192. global puzzle
193. global moves
194. zero\_ix = list(state).index(0)
195. actions = puzzle.actions(state)
196. current\_action = ''
197. i\_diff = index // 3 - zero\_ix // 3
198. j\_diff = index % 3 - zero\_ix % 3
199. if i\_diff == 1:
200. current\_action += 'DOWN'
201. elif i\_diff == -1:
202. current\_action += 'UP'
203. if j\_diff == 1:
204. current\_action += 'RIGHT'
205. elif j\_diff == -1:
206. current\_action += 'LEFT'
207. if abs(i\_diff) + abs(j\_diff) != 1:
208. current\_action = ''
209. if current\_action in actions:
210. b[zero\_ix].grid\_forget()
211. b[zero\_ix] = Button(root, width=size,height=size,command=partial(exchange, zero\_ix),bg=bgbtn,fg=ctxt,image=tk\_images[state[index]])
212. b[zero\_ix].grid(row=zero\_ix // 3, column=zero\_ix % 3,pady=distanceBtn,padx=distanceBtn)
213. b[index].grid\_forget()
214. b[index] = Button(root,width=size,height=size,command=partial(exchange, index),bg=bgbtn,fg=ctxt,image=tk\_images[0])
215. b[index].grid(row=index // 3, column=index % 3, pady=distanceBtn,padx=distanceBtn)
216. state[zero\_ix], state[index] = state[index], state[zero\_ix]
217. puzzle = EightPuzzle(tuple(state))
218. def create\_buttons():
219. """Creates dynamic buttons"""
220. *# TODO: Find a way to use grid\_forget() with a for loop for initialization*
221. b[0] = Button(root,  width=size,height=size,command=partial(exchange, 0),bg=bgbtn,fg=ctxt,image=f'{tk\_images[state[0]]}' if state[0] != 0 else tk\_images[0])
222. b[1] = Button(root,  width=size,height=size,command=partial(exchange, 1),bg=bgbtn,fg=ctxt,image=f'{tk\_images[state[1]]}' if state[1] != 0 else tk\_images[0])
223. b[2] = Button(root,width=size,height=size,command=partial(exchange, 2),bg=bgbtn,fg=ctxt,image=f'{tk\_images[state[2]]}' if state[2] != 0 else tk\_images[0])
224. b[3] = Button(root,width=size,height=size,command=partial(exchange, 3),bg=bgbtn,fg=ctxt,image=f'{tk\_images[state[3]]}' if state[3] != 0 else tk\_images[0])
225. b[4] = Button(root,width=size,height=size,command=partial(exchange, 4),bg=bgbtn,fg=ctxt,image=f'{tk\_images[state[4]]}' if state[4] != 0 else tk\_images[0])
226. b[5] = Button(root,width=size,height=size,command=partial(exchange, 5),bg=bgbtn,fg=ctxt,image=f'{tk\_images[state[5]]}' if state[5] != 0 else tk\_images[0])
227. b[6] = Button(root,width=size,height=size,command=partial(exchange, 6),bg=bgbtn,fg=ctxt,image=f'{tk\_images[state[6]]}' if state[6] != 0 else tk\_images[0])
228. b[7] = Button(root,width=size,height=size, command=partial(exchange, 7),bg=bgbtn,fg=ctxt,image=f'{tk\_images[state[7]]}' if state[7] != 0 else tk\_images[0])
229. b[8] = Button(root,width=size,height=size, command=partial(exchange, 8),bg=bgbtn,fg=ctxt,image=f'{tk\_images[state[8]]}' if state[8] != 0 else tk\_images[0])
231. b[0].grid(row=0, column=0,pady=distanceBtn,padx=distanceBtn)
232. b[1].grid(row=0, column=1,pady=distanceBtn,padx=distanceBtn)
233. b[2].grid(row=0, column=2,pady=distanceBtn,padx=distanceBtn)
234. b[3].grid(row=1, column=0,pady=distanceBtn,padx=distanceBtn)
235. b[4].grid(row=1, column=1,pady=distanceBtn,padx=distanceBtn)
236. b[5].grid(row=1, column=2,pady=distanceBtn,padx=distanceBtn)
237. b[6].grid(row=2, column=0,pady=distanceBtn,padx=distanceBtn)
238. b[7].grid(row=2, column=1,pady=distanceBtn,padx=distanceBtn)
239. b[8].grid(row=2, column=2,pady=distanceBtn,padx=distanceBtn)
241. def create\_static\_buttons():
242. """Creates scramble and solve buttons"""
243. btnbg = "#1877f2"
244. btnbg1 = "#36a420"
245. scramble\_btn = Button(root, text='Scramble', font=(ffm, 20, 'bold'), width=8, command=partial(init),bg="#fff",relief="solid",
246. background="#fff",
247. foreground="#135887",
248. activebackground="#5faee4",
249. activeforeground="#0e4163",
250. highlightbackground="GREEN",
251. highlightcolor='BLACK',
252. border=0,
253. cursor='hand1')
254. scramble\_btn.grid(row=0, column=5, ipady=20,padx=30, pady=10)
255. solve\_btn = Button(root, text='Solve', font=(ffm, 20, 'bold'), width=8, command=partial(solve\_steps),
256. borderwidth=2,bg="#fff",relief="solid",
257. background="#fff",
258. foreground="#135887",
259. activebackground="#5faee4",
260. activeforeground="#0e4163",
261. highlightbackground="GREEN",
262. highlightcolor='BLACK',
263. border=1,
264. cursor='hand1',
265. )
266. solve\_btn.grid(row=1, column=5, ipady=20,padx=30, pady=10)
267. def init():
268. """Calls necessary functions"""
269. global state
270. global solution
271. state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
272. scramble()
273. create\_buttons()
274. create\_static\_buttons()
275. init()
276. root.mainloop()
277. *# %%*

## **LẬP TRÌNH GIAO DIỆN GAME SOKOBAN**

### Giới thiệu chung về chủ đề

#### Mô tả yêu cầu của chủ đề

+ Lập trình Game sokoban bằng thư viện giao diện NumPy, os, colorama, pygame

+ Gồm 3 phần (Part) cơ bản sau

. Thư mục chứa thuật toán, các class hỗ trợ khai báo từng đối tượng cụ thể, các module hỗ trợ chạy thuật toán:

. Hiển thị giao diện cho người chơi.

. Tiến hành chạy từng thuật toán để giải quyết vấn đề.

+ Sokoban là một trò chơi thuộc thể loại giải đố, trong đó người chơi phải đẩy một số thùng hàng (khối vuông) vượt qua chướng ngại vật để đến được các đích cho sẵn). Ngoài ra:

. Gameplay là một bảng dạng hình vuông (map)

. Người chơi chỉ được đẩy một ô vuông một lần, không được kéo chúng

#### Phân tích yều cầu của chủ đề

+ Chủ đề lập trình python này là cơ sở của các ứng dụng về AI, ML, Deep ML và tập trung nhiều vào các thuật toán.

+ Ứng dụng các thuật toán thường dùng trong AI này vào giải quyết vấn đề cụ thể qua từng ma trận khác nhau từ ma trận ban đầu cho đến khi các hố đen được lấp đầy.

### Các nền tảng kỹ thuật

#### Nền tảng kỹ thuật cơ bản của giao diện

Thư viện Sử dụng:

* numpy: Được sử dụng để thực hiện các phép toán số học và xử lý mảng.
* os: Được sử dụng để tương tác với hệ điều hành, như đọc tệp và duyệt thư mục.
* colorama: Được sử dụng để thêm màu vào đầu ra trên cửa sổ console.
* pygame: Được sử dụng để tạo giao diện trò chơi Sokoban.
* Thư mục Sources: gồm các class hỗ trợ cho game được gộp thành một package với:
  + File Algorithm.py chứa các thuật toán hỗ trợ cho game như A\*, BFS, DFS, UCS, …
  + File support\_function.py chứa các hàm hỗ trợ như kiểm tra chiến thắng, kiểm tra thua cuộc, kiểm tra vị trí góc, tìm vị trí và class state để chứa trạng thái và các kết quả của trạng thái hiện tại.
  + File mainGui.py để chứa phần giao diện trình bày bằng pygame để lấy các Testcases, vị trí chứa hố Checkpoints, tải các hình ảnh game vào giao diện, vẽ map và chơi game.

Khởi Tạo và Đường Dẫn:

* Testcases và checkpoints được tải từ các thư mục cụ thể (Testcases và Checkpoints).
* Đoạn mã đọc các bảng Sokoban và điểm kiểm tra từ các tệp văn bản và định dạng chúng theo cách thích hợp.

Khởi Tạo Pygame:

* Pygame được khởi tạo, và các tài nguyên khác nhau (hình ảnh) được tải cho giao diện trò chơi.
* Đoạn mã thiết lập cửa sổ Pygame, đồng hồ và định nghĩa màu sắc và hình ảnh cho các yếu tố trò chơi khác nhau.

Các Cảnh Trò Chơi:

* Đoạn mã định nghĩa các cảnh như "khởi tạo," "đang tải," "đang chơi," và "kết thúc" để quản lý các trạng thái khác nhau của trò chơi.
* Hàm sokoban kiểm soát vòng lặp chính của trò chơi và chuyển đổi giữa các cảnh.

Lựa Chọn Thuật Toán:

* Người chơi có thể chọn các thuật toán khác nhau để giải bài toán Sokoban, chẳng hạn như BFS, DFS, Greedy, Iterative Deepening Search, UCS, A\*, và chơi thủ công.

Hiển Thị Trò Chơi:

* Hàm renderMap chịu trách nhiệm vẽ bản đồ trò chơi Sokoban trên cửa sổ Pygame.

Xử Lý Sự Kiện:

* Đoạn mã bắt các sự kiện bàn phím để điều khiển trò chơi, chẳng hạn như lựa chọn bản đồ, chọn thuật toán và điều hướng qua các cảnh.

Thời Gian Chờ:

* Có một thời gian chờ được đặt cho các thuật toán (1800 giây).

Hàm Chính:

* Đoạn mã kết thúc với việc gọi hàm main, bắt đầu trò chơi Sokoban.

#### Nền tảng, kĩ thuật cơ bản của các hàm hỗ trợ

\* class State():

init(self, board, state\_parent, list\_check\_point, algorithm=None): Khởi tạo một đối tượng state với trạng thái hiện tại của bảng, trạng thái cha, danh sách các điểm kiểm tra, và thuật toán sử dụng (mặc định là None).

get\_line(self): Trả về một danh sách các trạng thái từ trạng thái hiện tại đến trạng thái ban đầu (dùng để backtrack).

compute\_heuristic(self): Tính toán giá trị hàm heuristic cho thuật toán A\*, Greedy, hoặc UCS, tùy thuộc vào giá trị của biến algorithm.

\_\_gt\_\_(self, other): So sánh hai đối tượng state dựa trên giá trị heuristic để sử dụng trong Priority Queue.

\_\_lt\_\_(self, other): So sánh hai đối tượng state dựa trên giá trị heuristic để sử dụng trong Priority Queue.

check\_win(board, list\_check\_point): Kiểm tra xem trạng thái hiện tại của bảng có phải là trạng thái kết thúc (tất cả các điểm kiểm tra đều được che phủ bởi các hộp) hay không.

assign\_matrix(board): Sao chép ma trận hiện tại.

find\_position\_player(board): Tìm vị trí của người chơi (@) trên bảng.

compare\_matrix(board\_A, board\_B): So sánh hai ma trận xem chúng có giống nhau hay không.

is\_board\_exist(board, list\_state): Kiểm tra xem một trạng thái đã tồn tại trong danh sách trạng thái đã duyệt hay chưa.

is\_box\_on\_check\_point(box, list\_check\_point): Kiểm tra xem một hộp có đang ở trên một điểm kiểm tra không.

check\_in\_corner(board, x, y, list\_check\_point): Kiểm tra xem một hộp có bị kẹt ở góc không.

find\_boxes\_position(board): Tìm tất cả vị trí của các hộp ($) trên bảng.

is\_box\_can\_be\_moved(board, box\_position): Kiểm tra xem một hộp có thể di chuyển ít nhất 1 hướng không.

is\_all\_boxes\_stuck(board, list\_check\_point): Kiểm tra xem tất cả các hộp có bị kẹt hay không.

is\_board\_can\_not\_win(board, list\_check\_point): Kiểm tra xem bảng có thể không thể chiến thắng hay không (hộp kẹt ở góc).

get\_next\_pos(board, cur\_pos): Tìm các vị trí có thể di chuyển tiếp theo từ vị trí hiện tại của người chơi.

move(board, next\_pos, cur\_pos, list\_check\_point): Di chuyển người chơi đến một vị trí mới trên bảng và cập nhật bảng sau mỗi bước di chuyển.

find\_list\_check\_point(board): Tìm tất cả các điểm kiểm tra (%) trên bảng.

=> Các hàm này cung cấp các chức năng cơ bản để thực hiện giải thuật tìm đường đi trong trò chơi Sokoban, sử dụng các thuật toán như A\*, Greedy, và UCS.

#### Định nghĩa, hiệu suất của các thuật toán

**BFS (Breadth-First Search):**

- Thuật toán tìm kiếm theo chiều rộng.

- Sử dụng hàng đợi (queue) để duyệt qua từng trạng thái tiếp theo một cách tuần tự.

- Dừng khi tìm thấy trạng thái mục tiêu hoặc hết thời gian cho phép (TIME\_OUT).

- Thời gian chạy được đo bằng cách sử dụng module time.

**DFS (Depth-First Search):**

- Thuật toán tìm kiếm theo chiều sâu.

- Sử dụng ngăn xếp (stack) để duyệt qua các trạng thái ở chiều sâu.

- Dừng khi tìm thấy trạng thái mục tiêu hoặc hết thời gian cho phép.

- Cũng sử dụng module time để đo thời gian chạy.

**A Search:\***

- Thuật toán tìm kiếm thông minh kết hợp giữa tìm kiếm theo chiều rộng và hàm heuristic để đánh giá chất lượng của các trạng thái.

- Sử dụng hàng đợi ưu tiên (PriorityQueue) với giá trị ưu tiên là tổng chi phí đến trạng thái hiện tại và ước lượng chi phí còn lại.

- Dừng khi tìm thấy trạng thái mục tiêu hoặc hết thời gian cho phép.

**Greedy Search:**

- Thuật toán tìm kiếm theo chiều rộng sử dụng hàm heuristic nhưng không tính chi phí thực sự.

- Sử dụng hàng đợi ưu tiên với giá trị ưu tiên là giá trị heuristic.

- Dừng khi tìm thấy trạng thái mục tiêu hoặc hết thời gian cho phép.

**Uniform Cost Search:**

- Thuật toán tìm kiếm theo chiều rộng sử dụng chi phí thực sự để ưu tiên trạng thái.

- Sử dụng hàng đợi ưu tiên với giá trị ưu tiên là chi phí thực sự.

- Dừng khi tìm thấy trạng thái mục tiêu hoặc hết thời gian cho phép.

**IDS (Iterative Deepening Search):**

- Thuật toán tìm kiếm theo chiều sâu với độ sâu tăng dần.

- Lặp qua tất cả các độ sâu từ 1 đến vô cùng cho đến khi tìm thấy trạng thái mục tiêu hoặc hết thời gian cho phép.

Ngoài ra, các hàm trong mã nguồn được thiết kế để kiểm soát và đo lường hiệu suất của từng thuật toán bằng cách sử dụng biến timeRun và num\_states\_visited. Biến timeRun lưu trữ thời gian chạy của thuật toán và num\_states\_visited lưu trữ số lượng trạng thái đã kiểm tra trong quá trình tìm kiếm.

#### Nền tảng kỹ thuật cơ bản của thuật toán

\* Các Kỹ Thuật và Thành Phần Chung:

- Biểu Diễn Trạng Thái (state class): Lớp state biểu diễn một trạng thái cụ thể trong bài toán Sokoban, lưu trữ cấu hình bảng hiện tại, trạng thái cha, chi phí, giá trị heuristic, các điểm kiểm tra và loại thuật toán.

- Phương thức get\_line đệ quy quay lại trạng thái ban đầu, tạo ra một danh sách các trạng thái từ đầu đến trạng thái hiện tại.

- Hàm Heuristic: Phương thức compute\_heuristic tính toán giá trị heuristic cho mỗi trạng thái dựa trên thuật toán được chọn (A\*, Greedy, UCS). Giá trị heuristic được sử dụng để sắp xếp ưu tiên trong hàng đợi ưu tiên của A\* và Greedy.

- Hàng Đợi Ưu Tiên (PriorityQueue): Các thuật toán sử dụng hàng đợi ưu tiên để quản lý trạng thái dựa trên giá trị heuristic hoặc chi phí.

\* Các Hàm Thao Tác Bảng: Các hàm khác nhau được định nghĩa để thao tác bảng, chẳng hạn như di chuyển người chơi, kiểm tra chiến thắng, tìm vị trí của người chơi, so sánh bảng và kiểm tra hộp bị kẹt.

\* Các Thuật Toán Đã Triển Khai:

- BFS (Tìm Kiếm Theo Chiều Rộng): Sử dụng một hàng đợi để khám phá trạng thái theo chiều rộng. Đảm bảo rằng tất cả các trạng thái ở một độ sâu cụ thể được khám phá trước khi chuyển sang độ sâu tiếp theo.

- DFS (Tìm Kiếm Theo Chiều Sâu): Sử dụng một ngăn xếp để khám phá trạng thái theo chiều sâu.

- A\* Search: Sử dụng hàng đợi ưu tiên dựa trên tổng chi phí và giá trị heuristic để sắp xếp trạng thái. Hàm heuristic tính toán khoảng cách Manhattan giữa các hộp và các điểm kiểm tra tương ứng.

- Greedy Search: Tương tự như A\* nhưng chỉ xem xét giá trị heuristic mà không có thành phần chi phí.

- Uniform Cost Search (UCS): Biến thể của A\* nơi chỉ chi phí được xem xét để sắp xếp trạng thái.

- Iterative Deepening DFS (IDS): Lặp lại áp dụng DFS với giới hạn độ sâu tăng dần cho đến khi tìm thấy giải pháp.

\* Xử Lý Thời Gian Chờ:

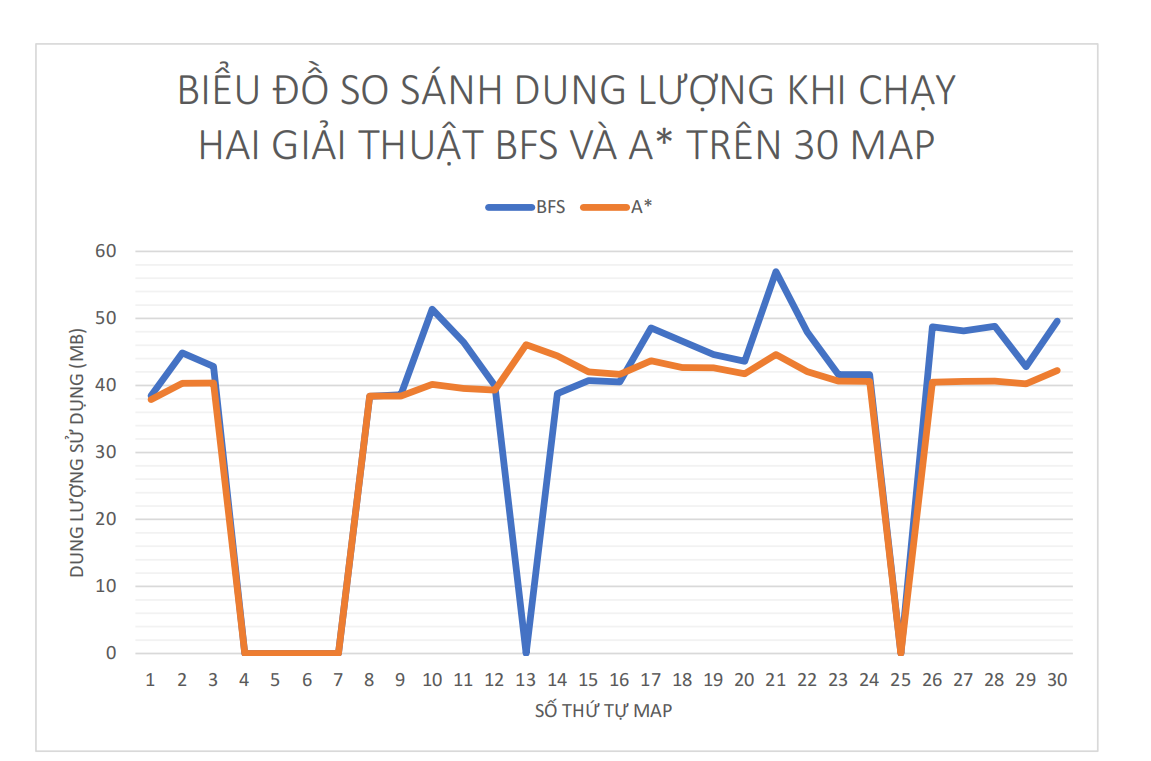
- Các thuật toán bao gồm một cơ chế thời gian chờ (spf.TIME\_OUT) để ngăn chặn việc thực thi kéo dài.

- Nếu thời gian thực thi vượt quá thời gian chờ được chỉ định, thì sẽ tắt ứng dụng.

#### So sánh biểu đồ ứng dụng của 2 thuật toán

\* Nhóm chúng em lấy đại diện cho 2 thuật toán thường chạy ra kết quả tối ưu nhất.

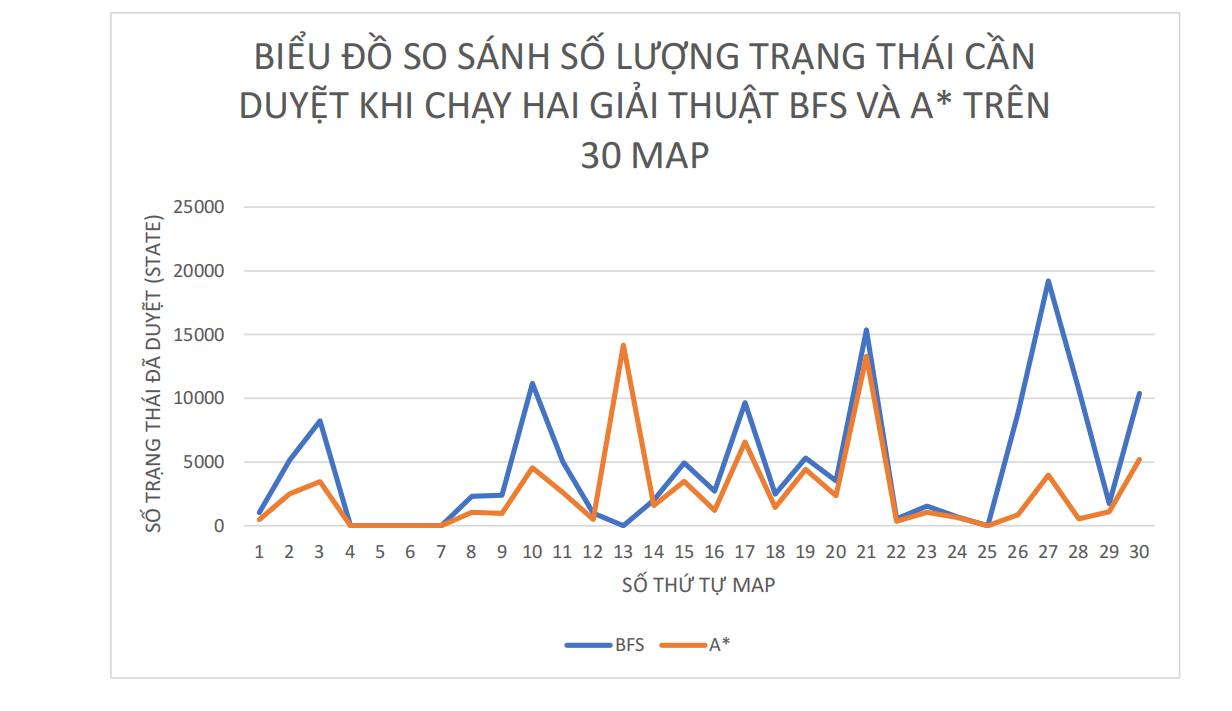
- Biểu đồ so sánh dung lượng khi chạy hai giải thuật BFS và A\* trên 30 map:



=> Dung lượng: Theo đồ thị, ta thấy 90% BFS chiếm nhiều dung lượng hơn A\*.

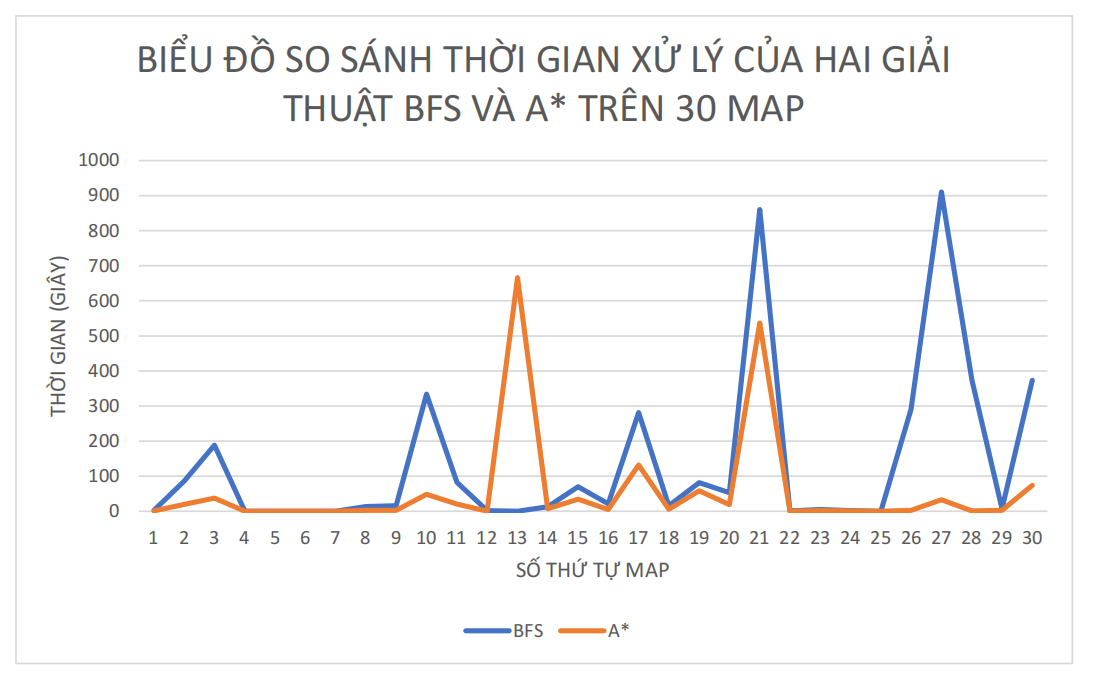
=> Timeout: Đa phần cả hai giải thuật đều có những map không thể giải.

- Biểu đồ so sánh số lượng trạng thái cần duyệt khi chạy hai giải thuật BFS và A\* trên 30 map:



=> Số lượng state: Theo thống kê, A\* luôn cho ra số state duyệt ít hơn BFS.

- Biểu đồ so sánh thời gian xử lí của hai giải thuật BFS và A\* trên 30 map:



=> Thời gian: Phần lớn A\* tối ưu hơn BFS về mọi mặt, kể cả thời gian.

#### Hệ thống thư việc sử dụng

- Trong file chính:

# Nạp folder chứa SokobanGame để chạy file main

sources\_path = os.path.join(os.path.dirname(\_\_file\_\_), 'SokobanGame', 'Sources')

sys.path.append(sources\_path)

def sokoban():

    from SokobanGame.Sources import mainGui

    mainGui.main()

play\_music\_bg()

- Trong file hỗ trợ thuật toán (Algorithm.py):

import support\_function as spf

import time

from queue import PriorityQueue

- Trong file hỗ trợ giao diện sokoban (mainGui.py)

import numpy as np

import os

from colorama import Fore

from colorama import Style

from copy import deepcopy

import pygame

from pygame.constants import KEYDOWN

import Algorithm as Algo

import time

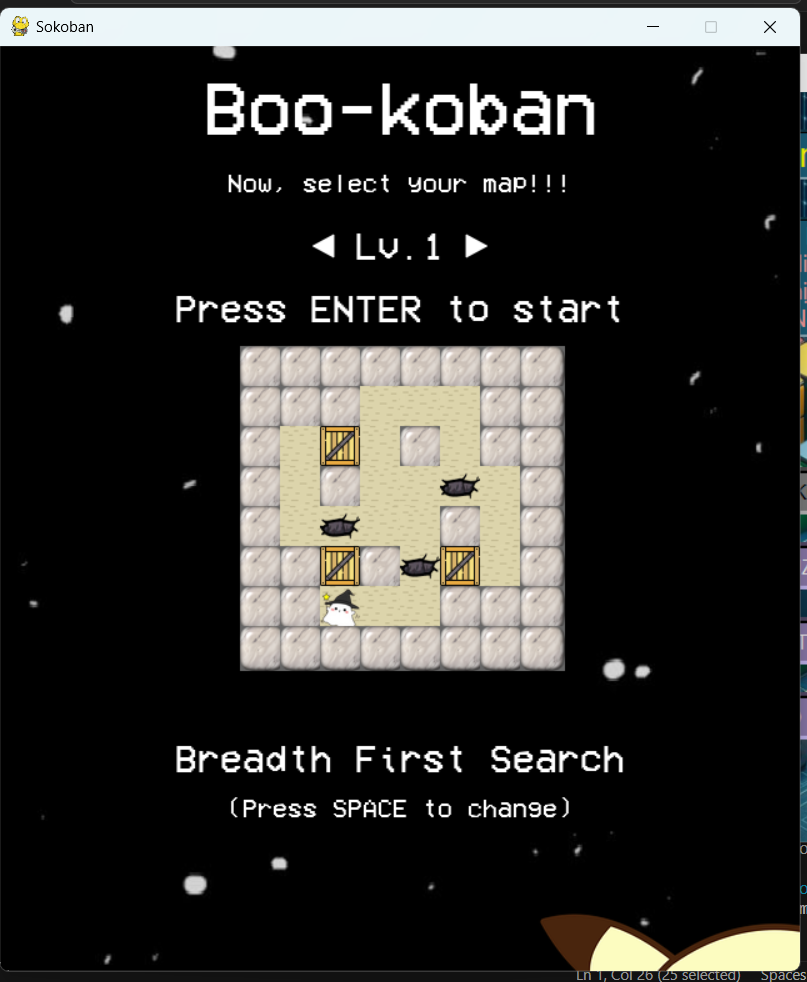
import support\_function as spf

- Trong file hỗ trợ các hàm (support\_function.py)

from copy import deepcopy

#### Thiết kế giao diện.

- Giao diện ban đầu:



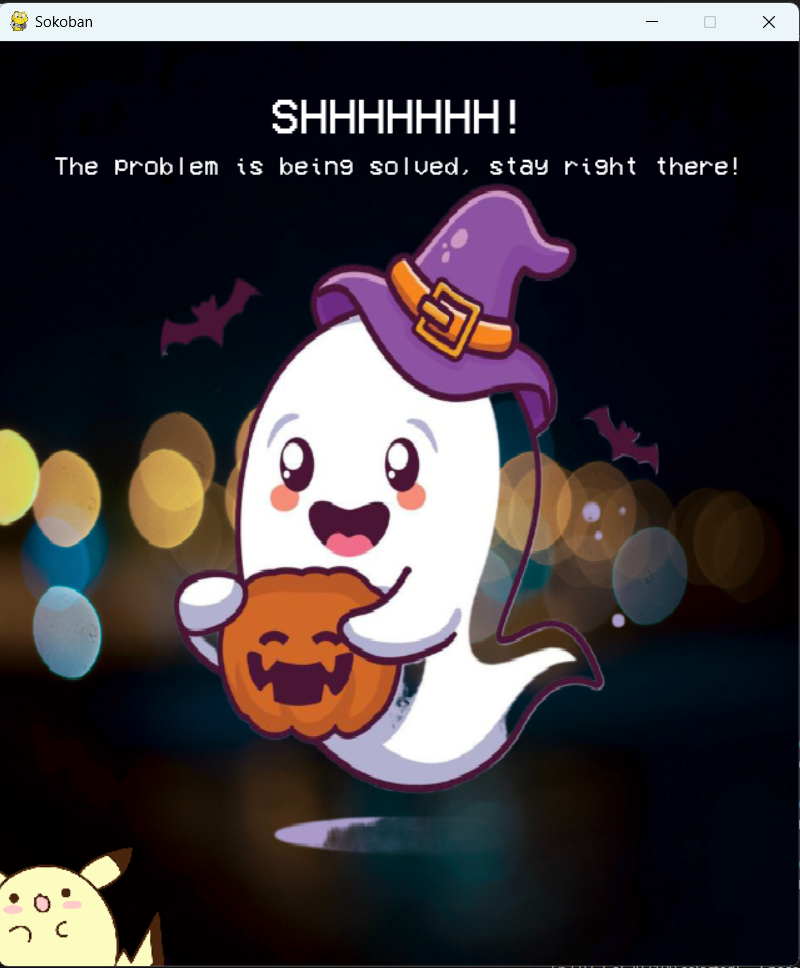
+ Nút SPACE (trên bàn phím): dùng để chuyển đổi qua lại giữa các thuật toán cài đặt và chế độ người chơi

+ Breadth First Search: tương trưng cho các thuật toán đã viết.

+ Chọn nút qua lại trên bàn phím (< >): là để chuyển qua lại giữa các map dựa trên Testcases và map sẽ được hiện ra bên dưới, sẽ có cấp độ cho từng map.

+ Nút ENTER: dùng để bắt đầu chạy game dựa trên thuật toán đã chọn ở dưới.

- Giao diện đang trong quá trình chạy thuật toán:



=> Chờ đến khi chạy xong thuật toán sẽ tự chuyển đổi giao diện.

- Giao diện đang chạy thuật toán tương ứng:  


=> Chờ đến khi các bước chạy đến kết quả hoàn thành sẽ cho ta các thông tin thông kê.

- Giao diện hoàn tất thuật toán:  


=> Ấn phím ENTER để tiếp tục chạy giao diện ban đầu để chọn map và thư mục.

- Giao diện người chơi:



=> Các nút điều khiển được hiện lên giao diện.

- Giao diện không tìm thấy giải pháp:



=> Ấn ENTER để tiếp tục cuộc chơi.

Phần 4: KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN CỦA ĐỒ ÁN HỌC PHẦN

## **4.1 KẾT LUẬN**

### 4.1.1 Những kết quả đạt được

ĐỒ ÁN HỌC PHẦN đã thực hiện việc lập trình dùng Python cho phân tích, bao gồm:

+ Giao diện đồ họa: GUI.  
+ Lập trình game đơn giản Sukoban.

+ Lập trình game đơn giản 8 - puzzle

### 4.1.2 Hạn chế.

Vì thời gian có hạn nên chưa liên kết voice assistant để điều khiển các chức năng khác ngoài EDA, chưa thêm nhiều chức năng hơn cho từng tác vụ và phân tích dữ liệu EDA còn thiếu sót. Các thao tác xử lí còn ít, chưa đa dạng, thuật toán game 8 - puzzle chưa hoàn toàn tối ưu.

## **4.2 HƯỚNG PHÁT TRIỂN**

### 4.2.1 Hướng khắc phục các hạn chế

Đồ án sẽ tiếp tục thực hiện:

* Liên kết voice assistant để ra lệnh điều khiển các giao diện khác ngoài EDA.
* Tạo ra cách chơi mới mẻ độc đáo hơn cho game Flappy Bird.
* Thêm nhiều chức năng cho voice assistant cũng như xử lý ảnh-video.
* Tăng thêm nhiều thuật toán với khả năng tối ưu cao hơn về thời gian chạy cũng như hiệu suất.

### 4.2.2 Hướng mở rộng ĐỒ ÁN HỌC PHẦN

Sẽ tiếp tục mở rộng sản phẩm đề tài với các ứng dụng thực tiễn.

DANH MỤC TÀI LIỆU THAM KHẢO

1. Tài liệu học tập Lập trình Python
2. Trang DataSet Kaggle dữ liệu EDA: <https://www.kaggle.com/datasets/ajaypalsinghlo/world-happiness-report-2021?select=world-happiness-report.csv>
3. Sách hỗ trợ AI: Russell 2016 Artificial intelligence a modern approach
4. Wikipedia - Tìm kiếm theo chiều sâu: <https://vi.wikipedia.org/wiki/T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u>
5. Wikipedia - Giải thuật tìm kiếm A\*: [https://vi.wikipedia.org/wiki/Gi%E1%BA%A3i\_thu%E1%BA%ADt\_t%C3%ACm\_ki%E1%BA%BFm\_A\*](https://vi.wikipedia.org/wiki/Gi%E1%BA%A3i_thu%E1%BA%ADt_t%C3%ACm_ki%E1%BA%BFm_A*)
6. Analytics Vidhya, Sriniketh J, August 10th, 2023, Uninformed Search Algorithms: Exploring New Possibilities (Updated 2023): <https://www.analyticsvidhya.com/blog/2021/02/uninformed-search-algorithms-in-ai/#:~:text=Uniform%20Cost%20Search%20Algorithm%20(UCS),-Uniform%20Cost%20Search&text=This%20algorithm%20is%20mainly%20used,node%20to%20the%20goal%20node>
7. Educative, What is Iterative Deepening Search?: <https://www.educative.io/answers/what-is-iterative-deepening-search>