HO CHI MINH CITY UNILVERSITY OF TECHNOLOGY AND EDUCATION



ĐỒ ÁN CUỐI KÌ

Môn: Trí Tuệ Nhân Tạo (Artificial Intelligence).

Giảng viên: Trần Nhật Quang

Nhóm 11

**Thành viên** :

17110008 Hồ Ngọc Đình Châu

17110195 Nguyễn Thị Bích Nhàn

Thủ đức , ngày 24 tháng 6 năm 2020

Mục Lục

[**A.** **Tìm hiểu về đạo văn** 5](#_Toc43889066)

[**1. Đạo văn là gì ?** 5](#_Toc43889067)

[**2. Những điều nên và không nên làm.** 5](#_Toc43889068)

[**B.** **Các cụm từ viết tắt** 6](#_Toc43889069)

[**C.** **Giới thiệu về Artificial Intelligence** 6](#_Toc43889070)

[**1.** **AI là gì?** 6](#_Toc43889071)

[**2.** **AI trong ngôn ngữ và giao tiếp** 6](#_Toc43889072)

[**3.** **AI in Computer Vision** 6](#_Toc43889073)

[**4.** **AI in Industry** 6](#_Toc43889074)

[**i.** **Other uses of AI** 7](#_Toc43889075)

[**ii.** **How far can AI go ?** 7](#_Toc43889076)

[**iii.** **Types of AI** 7](#_Toc43889077)

[**iv.** **AI good và AI bad** 7](#_Toc43889078)

[**5.** **Các trường phái đạo đức** 7](#_Toc43889079)

[**6.** **Tốt và xấu trong Bigdata** 7](#_Toc43889080)

[**7.** **Luật về xử lý dữ liệu** 8](#_Toc43889081)

[**8.** **Phân tích IRAC** 8](#_Toc43889082)

[**9.** **Đặt vấn đề : Lỗi AI hay con người** 9](#_Toc43889083)

[**10.** **XAI ( Explainable AI )** 9](#_Toc43889084)

[**11.** **Điểm qua điểm chính trong lịch sử AI** 9](#_Toc43889085)

[**12.** **Tiếp cận AI** 11](#_Toc43889086)

[**13.** **Lĩnh vực nguyên cứu AI** 11](#_Toc43889087)

[**14.** **Các khái niệm căn bản của AI** 11](#_Toc43889088)

[**15.** **Nguyên tắc thiết kế Reward** 12](#_Toc43889089)

[**16.** **Exploit và Explore** 12](#_Toc43889090)

[**17.** **Các loại môi trường (Types of environments)** 12](#_Toc43889091)

[ **Fully observable** 12](#_Toc43889092)

[ **Partially observable** 13](#_Toc43889093)

[ **Unobservable** 13](#_Toc43889094)

[**18.** **Các loại môi trường ( Types of environment) tiếp theo** 13](#_Toc43889095)

[ **Single agent** 13](#_Toc43889096)

[ **Multiagent** 13](#_Toc43889097)

[ **Episidic** 13](#_Toc43889098)

[ **Sequential** 13](#_Toc43889099)

[**19.** **State representations** 13](#_Toc43889100)

[ **Atomic\_nguyên tử** 14](#_Toc43889101)

[ **Factored** 14](#_Toc43889102)

[ **Structured** 14](#_Toc43889103)

[**20.** **PEAS mô tả agents** 14](#_Toc43889104)

[**21.** **Các loại agents cơ bản** 15](#_Toc43889105)

[ **Simple reflex agent ( agent phản xạ đơn giản)** 15](#_Toc43889106)

[ **Model-based reflex agents** 15](#_Toc43889107)

[ **Goal-based agents** 16](#_Toc43889108)

[ **Utility-based agents** 16](#_Toc43889109)

[ **Learning agents** 16](#_Toc43889110)

[**22.** **Solving problems by searching** 16](#_Toc43889111)

[**23.** **Mô tả bài toán bằng Components** 17](#_Toc43889112)

[**D.** **Thuật toán trong AI.** 17](#_Toc43889113)

[**1.** **Searching for solutions** 17](#_Toc43889114)

[**2.** **Tree Search** 17](#_Toc43889115)

[**3.** **Vấn đề: Repeated states 🡪 Graph Search** 17](#_Toc43889116)

[**4.** **Developing search algorithms.** 18](#_Toc43889117)

[**5.** **Uninformed search** 18](#_Toc43889118)

[ **Breadth-first search (BFS)** 18](#_Toc43889119)

[ **Uniform-cost search algorithm (UCS)** 20](#_Toc43889120)

[ **So sánh những thuật toán** 21](#_Toc43889121)

[ **Depth-first search (DFS)** 21](#_Toc43889122)

[ **Depth-limited search(DLS)** 22](#_Toc43889123)

[ **Interative deepending search** 23](#_Toc43889124)

[ **So sánh giữa DFS và BFS** 24](#_Toc43889125)

[ **So sánh những thuật toán đã học** 24](#_Toc43889126)

[**6.** **Informed search algorithms** 24](#_Toc43889127)

[ **Evaluation function f(n)** 25](#_Toc43889128)

[ **Best-first search** 25](#_Toc43889129)

[ **A\* search** 25](#_Toc43889130)

[ **So sánh thuật toán** 26](#_Toc43889131)

[ **Generating heuristic functions** 26](#_Toc43889132)

[**7.** **Search in more complex environments** 26](#_Toc43889133)

[ **Local search** 26](#_Toc43889134)

[ **Hill-climbing search** 26](#_Toc43889135)

[ **Simulated annealing algorithm** 27](#_Toc43889136)

[ **The erratic vacuum world** 28](#_Toc43889137)

[Điển hình như máy hút bụi hư , 1 action của nó có thể hút nhưng lại vừa có thể thả bụi ra. 28](#_Toc43889138)

[ **AND – OR search tree** 28](#_Toc43889139)

[ **AND – OR graph search algorithm** 28](#_Toc43889140)

[ **Partially observable environments** 28](#_Toc43889141)

[ **Sensorless problems** 28](#_Toc43889142)

[ **Incremental belief-state search** 29](#_Toc43889143)

[**9.** **Searching with no transition model – online search** 29](#_Toc43889144)

[ **Offline và online** 29](#_Toc43889145)

[ **Lợi ích ủa Online search** 29](#_Toc43889146)

[ **Competitive ratio** 29](#_Toc43889147)

[ **Dead-en state** 29](#_Toc43889148)

[ **Online depth-first search agent** 30](#_Toc43889149)

[ **Tại sao lại sử dụng depth-first?** 30](#_Toc43889150)

[ **Online A\* search LRTA\*(Learning Real-Time A\*)algorithm** 30](#_Toc43889151)

[ **Constraint satisfaction problems(CSP)** 31](#_Toc43889152)

[ **Solve CSPs using Constrain Propagation.** 31](#_Toc43889153)

[ **Constraint propagation** 31](#_Toc43889154)

[ **Local consisitency** 31](#_Toc43889155)

[ **Node consistency** 31](#_Toc43889156)

[ **Arc consistency** 31](#_Toc43889157)

[ **AC-3 algorithm** 31](#_Toc43889158)

[ **Path consistency** 32](#_Toc43889159)

[ **K-consistency** 32](#_Toc43889160)

[ **Global constraints** 32](#_Toc43889161)

[ **Solve CSPs using Backtracking Search** 32](#_Toc43889162)

[ **Comments on backtracking search algorithm** 33](#_Toc43889163)

[ **Min-conflicts algorithm** 33](#_Toc43889164)

[ **Sdas** 33](#_Toc43889165)

[ **Dsad** 33](#_Toc43889166)

[ **sdas** 33](#_Toc43889167)

# **Tìm hiểu về đạo văn**

## **1. Đạo văn là gì ?**

Đạo văn là hành vi ăn cắp hay sử dụng sản phẩm và ý tưởng của ai đó mà không ghi rõ nguồn, hoặc coi như đó là sản phẩm của bản thân mình. Đạo văn là một hành động được coi là tối kỵ trong học tập và nghiên cứu, vì nó để lại những hậu quả khá nặng nề cho cả người đạo văn và xã hội. Hiện nay, đạo văn bị bài trừ gắt gao trên khắp thế giới. Sinh viên có hành vi đạo văn có thể phải chịu những hình phạt nặng nề như bị đuổi học, phạt tiền, thậm chí còn phải chịu trách nhiệm trước pháp luật.

Tại Việt Nam, đạo văn đang là một vấn nạn trong nhiều lĩnh vực như giáo dục, khoa học, xuất bản… Thế nhưng, chưa có một chương trình học thuật nào đào tạo bài bản cho học sinh về đạo văn, cũng như cách phòng tránh thói quen đạo văn ngay từ khi còn nhỏ.

## **2. Những điều nên và không nên làm.**

|  |  |
| --- | --- |
| Nên làm | Không nên làm |
| * Nắm rõ qui định trích nguồn. * Tập diễn đạt ý tưởng theo. nhiều cách khác nhau. * Nêu tên tác giả của ý tưởng đó. * Tìm hiểu kiến thức về bản quyền | * Thuê người khác viết bài luận * Cố tình biến ý tưởng, bài luận của người khác thành của mình * Tải và sử dụng một hay nhiều bài báo trên mạng mà chưa xin phép bản quyền * Sao chép phân bố, bố cục từ |

1. **Các cụm từ viết tắt**

|  |  |
| --- | --- |
| AI | Artificial Intelligence |
| BFS | Breadth-first search |
| UCS | Uniform-cost search algorithm |
| DLS | Depth-limited search |
| DFS | Depth-first search |
| DLS | Depth-limited search |
|  |  |
|  |  |

# **Giới thiệu về Artificial Intelligence**

## **AI là gì?**

AI ( Artificial Intelligence ) là công nghệ mô phỏng các quá trình suy nghĩ và học tập của con người hay nói cách khác AI là trí thông minh được thể hiện bằng máy móc.

## **AI trong ngôn ngữ và giao tiếp**

* AI phải hiểu được những ngôn ngữ tự nhiên ( Giống như con người nói chuyện với nhau).
* Dịch từ ngôn ngữ này sang ngôn ngữ khác.

## **AI in Computer Vision**

* AI có thể hiểu qua hình ảnh

VD: Chụp một hình ảnh nào đó nó có thể biết mình đang ở đâu hoặc là dịch văn bản qua hình ảnh

## **AI in Industry**

* Trong quy trình phun thuốc diệt cỏ, nếu AI phát hiện ra cây cỏ thì nó mới phun thuốc.

### **Other uses of AI**

* Vẽ hình ảnh như thật .
* Lấp đầy những khoảng trắng.
* Chơi game với con người.

### **How far can AI go ?**

Thông minh hơn con người.

### **Types of AI**

* Weak AI ( narrow ) : AI chỉ làm được một việc nhất định .
* General AI (strong) : AI làm được nhiều công việc như một con người thật sự.

### **AI good và AI bad**

|  |  |
| --- | --- |
| **BAD for others** | **GOOD for others** |
| Killing | Saving lives |
| Stealing | Helping others in need |
| Being lazy, drunk | Learning, working hard |
| Lying | Being honest |
| Having an extramarital relationship | Being faithful |

1. **Các trường phái đạo đức**

|  |  |  |
| --- | --- | --- |
| **Virtue ethics** | **Deontology** | **Consequentialism** |
| * Tuân theo những cái đức hạnh | * Tuân theo những quy luật ( rules) | * Ảnh hưởng tới mặc kết quả |

1. **Tốt và xấu trong Bigdata**

Cũng là 1 chức năng những nó cũng sẽ có 2 mặt trong việc sử dụng.

|  |  |  |
| --- | --- | --- |
| **Chức năng** | **Tốt** | **Chưa biết rõ tốt hay xấu** |
| Đọc biển số xe | * Nếu sử dụng chỉ đọc những chiếc xe vi phạm luật giao thông * Cảnh sát định vị những chiếc xe bị cướp | * Theo dõi người khác |
| Định diện khuôn mặt | * Phát hiện những tên khủng bố trong đám đông | * Nhận ra các thành viên trong 1 bức tranh 🡪 tùy thuộc vào những việc làm sau đó mà phân biệt tốt hay xấu |
| GPS | * Giúp dẫn đường, tránh kẹt xe,.. | * Theo dõi, quấy rối người khác |

1. **Luật về xử lý dữ liệu**

* Có rất nhiều bộ luật ở các nước nêu rõ về việc xử lý dữ liệu

Ở Châu Âu:

* Phân tích dữ liệu một cách công bằng (phân biệt màu da, giới tính)
* Khi muốn xử lý dữ liệu thì bạn phải xin phép chủ nhân của dữ liệu đó.
* Cá nhân của một cá thể phải có quyền truy xuất vào dữ liệu cá nhân của họ và chỉnh sửa nếu phát hiện sai.
* Phải có tính chịu trách nhiệm nếu việc xử lý dữ liệu ảnh hưởng mọi người.
* Và còn rất nhiều bộ luật khác để đảm bảo việc xử lý dữ liệu trở nên tốt hơn công bằng cho tất cả mọi người.

1. **Phân tích IRAC**

IRAC là phương thức để xét xem tình huống có vi phạm vào các luật lệ hay không.

I → Issue : Định nghĩ về vấn đề pháp lý

R → Rule : Tìm những bộ luật liên quan tới vấn đề

A → Application : Xem trong bộ luật xem vấn đề vi phạm chỗ nào trong bộ luật hay là không vi phạm.

C → Conclusion : Kết luận

1. **Đặt vấn đề : Lỗi AI hay con người**

Khi AI ngày càng phát triển → hầu như sẽ thay thế con người quyết định mọi thứ.

Vấn đề đặt ra : Nếu AI quyết định sai → Lỗi do ai ? ( AI hay con người)

VD: Một chiếc xe được điều khiển bởi AI và tài xế giao toàn quyền quyết định cho AI xử lý → hậu quả là xe do AI điều khiển đã gây ra tai nạn chết người

Ngay vào thời điểm ban đầu người tài xế đã sai khi giao toàn quyền quyết định lái chiếc xe cho AI. Dù đang chỉnh chế độ lái tự động nhưng tài xế vẫn không được lơ là khi tham gia giao thông.

Tuy nhiên, liệu AI có lỗi hay không ? Ta sẽ chia ra các trường hợp sau

* TH1: Nếu khi kiểm tra hệ thống gặp lỗi ( Do lập trình ) hoặc là hệ thống chưa hoàn hảo thì lỗi sẽ nằm ở các kĩ sư thiết kế hệ thống đó.
* TH2: Nếu phần mềm không gặp lỗi tuy nhiên hệ thống vẫn còn những hạn chế nhất định mà bên bán chiếc xe không cảnh báo trước cho người mua thì lúc này người có lỗi lại là công ty bán xe.

→ Vì vậy chưa thể kết luận bên nào sai khi chưa có những trường hợp cụ thể.

1. **XAI ( Explainable AI )**

Là một trong những dự án của DAPRA là những hệ thống machine-learning nó có thể giải thích các lý luận, đưa ra dự báo trong tương lai để có thể tránh đi những sai lầm không đáng có khi sử dụng AI.

Training Data → New Machine Learning Process → Explainable Model → Explanation Interface

→ Tuy nhiên dự án này vẫn đang trong quá trình nguyên cứu.

1. **Điểm qua điểm chính trong lịch sử AI**

|  |  |
| --- | --- |
| Năm | Sự kiện |
| 1956 | * Cụm từ trí tuệ nhân tạo lần đầu tiên được nói đến tại “Dự án nghiên cứu mùa hè về trí tuệ nhân tạo”. Với sự dẫn đầu bởi John McCarthy, hội nghị, trong đó xác định phạm vi và mục tiêu của AI, được coi là sự ra đời của trí tuệ nhân tạo như chúng ta biết ngày nay. * AI được coi như một ngành độc lập. |
| 1958 | * John McCarthy phát triển ngôn ngữ lập trình AI Lisp và xuất bản bài báo “Programs with Common Sense”. Bài viết đã đề xuất nhà tư vấn giả thuyết, một hệ thống AI hoàn chỉnh với khả năng học hỏi kinh nghiệm hiệu quả như con người. |
| 1959 | * Allen Newell, Herbert Simon và JC Shaw giải quyết vấn đề chung (GPS), một chương trình được thiết kế để bắt chước giải quyết vấn đề của con người. * Herbert Gelernter phát triển chương trình Định lý hình học. * Arthur Samuel đồng xu với thuật ngữ học máy khi còn ở IBM. * John McCarthy và Marvin Minsky đã tìm thấy Dự án Trí tuệ nhân tạo MIT. |
| 1963 | * John McCarthy bắt đầu Phòng thí nghiệm AI tại Stanford. |
| 1969 | * Các hệ thống chuyên gia thành công đầu tiên được phát triển trong DENDRAL, một chương trình XX và MYCIN, được thiết kế để chẩn đoán nhiễm trùng máu, được tạo ra tại Stanford. |
| 1980 | * Tập đoàn thiết bị kỹ thuật số phát triển R1 (còn được gọi là XCON), hệ thống chuyên gia thương mại thành công đầu tiên. Được thiết kế để định cấu hình các đơn đặt hàng cho các hệ thống máy tính mới, R1 khởi đầu sự bùng nổ đầu tư vào các hệ thống chuyên gia sẽ tồn tại trong phần lớn thập kỷ, kết thúc hiệu quả “Mùa đông AI” đầu tiên. |
| 2005 | * STANLEY, một chiếc xe tự lái, chiến thắng DARPA Grand Challenge. * Quân đội Hoa Kỳ bắt đầu đầu tư vào các robot tự hành như “Big Dog” của Boston Dynamic và “PackBot” của iRobot. |
| 2014 | * Google tạo ra chiếc xe tự lái đầu tiên để vượt qua bài kiểm tra lái xe của nhà nước. |
| 2016 | * AlphaGo của Google DeepMind đánh bại nhà vô địch thế giới cờ vây Lee Sedol. Sự phức tạp của trò chơi Trung Quốc cổ đại được coi là một trở ngại lớn để giải tỏa trong AI. |

1. **Tiếp cận AI**

|  |  |  |  |
| --- | --- | --- | --- |
| **Thinking humanly** | **Thinking rationally** | **Acting humanly** | **Acting rationally** |
| Xây dựng những trí tuệ giống như con người. | Xây dựng AI biết suy nghĩ giải quyết vấn đề bằng cách nào cho hiệu quả | Xây dựng hệ thống hoạt động như con người | Xây dựng hệ thống làm việc hiệu quả có thể bằng hoặc hơn con người |
| Quan tâm các tiến trình suy nghĩ , lập luận của AI. | | Quan tâm về output hay hành vi của AI đó. | |

1. **Lĩnh vực nguyên cứu AI**

* Philosophy
* Math : Logic
* Economics
* Meuro science
* Psychology
* Linguistics
* Computer engineering
* Control theory

1. **Các khái niệm căn bản của AI**

* Agents : đối tượng xử lý trí tuệ (giống như robot hay là một phần mềm)
* Environment : môi trường xung quanh Agents
* Actions : các hành động của Agents tác động vào Environment.
* Percepts : Các Agents có thể quan sát được khi tác động vào Environment
* Rewards : Cho Agets biết điều nó làm là tốt hay xấu.

1. **Nguyên tắc thiết kế Reward**

* Thường Rewards do con người thiết kế cho hệ thống**.**
* Phải có thưởng thì phải có phạt để tránh việc hao tốn tài nguyên hoặc mang những tác hại không muốn.
* Tùy loại hệ thống mà thiết kế làm sao cho phù hợp phục vụ đúng yêu cầu của người dùng. Tránh việc lãnh phí tài nguyên mà không thu được gì cả.
  + Qui tắc khi thiết kế reward systems:
* A general rule
  + Thiết kế agent hoạt động theo cách chúng ta muốn thay vì theo cách chúng ta nghĩ nó cần làm gì.
  + Bad behaviors → Bad results but **some** good behaviors **alone →** may not enough to get good result.

1. **Exploit và Explore**

|  |  |
| --- | --- |
| **Exploit** | **Explore** |
| Agents sẽ đứng im một chỗ để khai thác những điểm rewards | Agents sẽ đi khám phá mọi ngóc ngách cho đến khi nào tìm được nơi thu được nhiều rewards nhất. |

* Tùy theo trường hợp mà Agents sử dụng Exploit hay Explore.
* Khi lần đầu tham gia một hoạt động nào đó thì Agents ưu tiên sẽ sử dụng Explore để tìm ra nơi có thể khai thác được nhiều rewards nhất. Sau khi đã tìm ra được thì Agents sẽ sử dụng Exploit đề tối ưu hóa việc chiếm lấy rewards nhiều hết mức có thể.

1. **Các loại môi trường (Types of environments)**

Để có thể đánh giá thử xem Agents có thể hiểu rõ môi trường tới đâu. Môi trường được chia làm 3 phần:

* **Fully observable**

Agents có thể biết được đầy đủ về môi trường xung quanh nó. Nói cách khác Agents được trang bị đầy đủ mọi loại thiết bị, cảm biến để có thể nhận biết được môi trường nó sẽ và đang làm việc.

* Tuy nhiên trong thực thế rất khó để Agents có thể đạt tới Fully observable.

### **Partially observable**

Agents chỉ biết về một phần nào đó về môi trường xung quanh nó mà thôi. Agents được không được trang bị những thứ cần thiết không đủ chất lượng để agents có thể nhận biết về mọi thứ về môi trường xung quanh nó.

* Trong thực tế , trường hợp này thường xảy ra.

### **Unobservable**

Agents hoàn toàn không biết gì về môi trường xung quanh nó. Trong thực thế Unobservable vẫn có thể giải quyết được một số việc nhưng sẽ kém hiệu quả rất là nhiều so với Fully observable và Partially obserbable.

1. **Các loại môi trường ( Types of environment) tiếp theo**

### **Single agent**

Trong môi trường hệ thống chỉ có một agent hoạt động độc lập

Vd: game ô chữ, game giải đố

### **Multiagent**

Trong môi trường này có nhiều agent hoạt động với nhau.

Vd : Chơi cờ ,game đối kháng,xe tự lái

* **Episidic**

Episidic sẽ tách biệt ra với nhau;

* Trở về trạng thái ban đầu sau khi thực hiện một episode.
* Episode trước sẽ không ảnh hưởng tới episode sau.
* **Sequential**

Không có trạng thái bắt đầu kết thúc

1. **State representations**

### **Atomic\_nguyên tử**

Dạng đơn giản nhất có thể hiểu đây giống như một black-box, chỉ có tên trạng thái chứ ngoài ra bên trong nó không có một chút thông tin gì cả. ( SA = State A).

Thường được dùng trong các thuật toán Search và Marup processes.

### **Factored**

Dạng này sẽ chi tiết hơn dạng trước. Nếu dạng trước bên trong trống rỗng thì dạng này có cả chi tiết (features).

Trời mưa:

Mây nhiều

Độ ẩm cao

Nhiệt độ thấp: 20 độ

Trời nắng:

Nhiệt độ cao 38 độ

State : trời mưa

Features : Mây nhiều , độ ẩm cao , nhiệt độ thấp ,vv..

Thường được dùng trong những thuộc toán Machine learning.

### **Structured**

Dạng là dạng cấu trúc, Factored cho ta thấy những features bên trong trạng thái còn nhưng chúng độc lập với nhau không có sự liên kết còn Structured thì các features sẽ liên kết chặc chẽ với nhau. Chỉ cần thay đổi một feature thì nó sẽ ra dạng feature khác.

Thường được dùng trong Relational Database, Natural language.

1. **PEAS mô tả agents**

Mô tả agents một cách chi tiết hơn

P 🡪 Performance (Reward system) : Đo thử xem khả năng thực thi của Agent này là tốt hay xấu

E 🡪 Environment : Thứ tác động với agent

A 🡪 Actuators : Bộ phận giúp agents có thể action được.

S 🡪 Sensors : Giúp agent hiểu được môi trường xung quanh (bộ phận cảm biến)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Agent Type** | **Performance** | **Environment** | **Actuators** | **Sensors** |
| Dạy tiếng anh | Tăng điểm thi tiếng Anh. | Nhóm sinh viên | Hiển thị màn hình các bài tập, các gợi ý, sửa bài tập | Bàn phím |
| Nhặt đồ vật | Tỉ lệ (%) bao nhiệu đồ vật được đặt đúng vào các thùng | Dây chuyền chuyển động trên đó có các đồ vậy, các thùng đựng | Cánh tay và bàn tay được kết nối | Máy quay,các cảm biến các góc độ. |

1. **Các loại agents cơ bản**

### **Simple reflex agent ( agent phản xạ đơn giản)**

Agent chỉ lựa chọn hành động duy nhất tại current percept. Không cần biết những các percept trước nó là gì.

|  |  |
| --- | --- |
| **Percept sequence** | **Action** |
| [A,Clean] | Right |
| [A,Clean], [A,Clean] | Right |
| [A,Clean], [A,Clean], [A,Dirty] | Suck |
| [A,Clean], [B,Clean] | Left |

Hạn chế : Simple reflex agent không hoạt động tốt tại mội trường partially observable

### **Model-based reflex agents**

* Agents nó sẽ xây dựng một cái mô hình của environment.
* Đối với loại agents này nó sẽ càng hiểu rõ hơn về environment nhờ thế nó có đoán được những các environment mà nó không nhìn thấy.

### **Goal-based agents**

* Các loại trước agents phản xạ theo những trạng thái (state) của environment. Tuy nhiên nếu chỉ như vậy thôi thì vẫn chưa đủ.
* Goal-based agents thì nó không chỉ dựa trên state của environment mà còn dựa theo mục tiêu của nó để nó đưa ra action cho hợp lý 🡪 loại agent sẽ hoạt động hiệu quả hơn

### **Utility-based agents**

* **Goals** nó chỉ cung cấp cho ta sự phân biệt về mặt nhị phân là “DONE” and “NOT DONE” thì nó không đủ để ta có thể tạo ra một agent chất lượng.
* **Utility** nó không chỉ quan tâm đên agents làm có được hay không còn quan tâm đến việc những agents hoạt động tốt như thế nào.

### **Learning agents**

* Nó bao gồm tất cả các loại agent trên và qua thời gian nó sẽ càng thông minh hơn, chất lượng và làm việc hiệu quả hơn.
* Hiện tại đây là những loại agents tốt nhất mà chúng ta có.
* Gồm 4 thành phần
* Performacve element: nhận vào percept và đưa ra action ( theo những rule mà nó học được).
* Critic: nhận vào percept và trả về feedback (xem agent có hoạt động đạt hiệu quả hay không).
* Learning element: Nhận feedback và cải thiện ( tạo ra những rule mới).
* Problem generator: Tạo ra explore (khám phá để tìm ra action tốt nhất)

1. **Solving problems by searching**

Xây dựng một agent cụ thể sử dụng thuật toán tìm kiếm:

* Là một dạng goal-based agent
* Sử dụng atomic representation

Mô tả bài toán:

* Goal: tập hợp những states mà agent muốn đạt đến.
* Solution: Giải pháp 🡪 chuỗi những action mà agent sẽ làm.
* (Optimal solution) 🡪 giải pháp tốt nhất để giải quyết cho 1 vấn đề nào đó.
* Search: quá trình tìm kiếm solution cho agent 🡪 giúp cho agent đạt được mục tiêu
* State space: tập hợp tất cả những state có thể.
* Level of abstraction: mức độ trừu tượng quá của action.

1. **Mô tả bài toán bằng Components**
   * + Initial state : trạng thái bắt đầu của agent.
     + Possible actions: Những các action sẽ sử dụng trong state đó KH: ACTION[state s].
     + Trangsition model: kết quả của 1 action trong 1 state. KH: RESULT(state s, action a).
     + Goal test: Phép thử state hiện tại có phải goal state hay không(Yes/No)
     + Cost: Tính xem 1 action nó sẽ tốn chi phi như thế nào ( càng thấp càng tốt).
2. **Thuật toán trong AI.**
   1. **Searching for solutions**

Gồm có 2 bước:

* Chọn một cái node/state
* Làm những action cho những state
  1. **Tree Search**

Trả về một solution(một chuỗi những action Initial state 🡪 goal) và khởi tạo frotier( tập hợp các lớp lá của node).

**Loop do**

**If** the frontier rỗng then **return** failire(thất bại)

* Chọn 1 leaf node và xóa nó khỏi frontier

**If** node chính là goal state then **return** the corresponding solution

**Nếu không phải goal state** 🡪 expand những node nó ra, them node đó vào frontier

* 1. **Vấn đề: Repeated states 🡪 Graph Search**

Trùng lặp vị trí state mà agent đã đi qua 🡪 dẫn tới vòng lặp vô tận trong thuật toán Tree Search. 🡪 Cần tìm ra một giải pháp để giải quyết nó

Graph Search chính là giải pháp đó.

Hàm Graph Search nó khác Tree Search là nó sẽ có thêm một khái niệm mới là explored set. Đây là tập hợp chứ những điểm mà agent đã đi qua đã được expand.

**Loop do**

**If** the frontier rỗng then **return** failire(thất bại)

* Chọn 1 leaf node và xóa nó khỏi frontier

**If n**ode chính là goal state then **return** the corresponding solution

* **Thêm node vào explored set.**

**Nếu không phải goad state** 🡪 expand những node nó ra, thêm node đó vào frontier.

* **Nếu mà node này không nằm trong frontier hoặc explored set 🡪 thì mới thêm vào frontier.**
  1. **Developing search algorithms.**

Chúng ta sẽ dựa vào 2 thuật toán trên : Three Search và Graph Search để tạo ra một thuật toán tìm kiếm cụ thể. Thuật toán tìm kiếm này có 2 loại;

* Uninformed search (tìm kiếm mà không có thông tin): Ta chỉ biết những problem.
* Informed search (tìm kiếm có thông tin): ngoài problem ta sẽ có thêm những thông tin khác để hỗ trợ agent làm việc hiệu quả hơn.
  1. **Uninformed search**
     + **Breadth-first search (BFS)**

Đây là thuật toán tìm kiếm theo chiều rộng.

* Bắt đầu từ root node rùi expaned nó.
* Sau đó ta sẽ expanded tất cả successors ( node con được mở rộng).
* Sau đó tiếp tục với tất cả their successors 🡪 end.

Một khái niệm mới là depth ( số action thực hiện để tới state đó).

Ta gọi node là n:

* n.STATE: state.
* n.PARENT: là node tạo ra nó.
* n.ACTION: action.
* n.PATH-COST: tổng cost tính từ lúc bắt đầu.

CHILD-NODE(problem,parent,action) 🡪 tìm ra node con

**Queues :** Là Array để chứa các element gồm 3 phép toán

EMPTY(queue): trả về true nếu rỗng

POP(queue): lấy ra 1 phần tử

INSERT(element,queue): chèn vào 1 phần tử

Có 3 loại QUEUES phổ biến

* FIFO queie ( first-in, first-out)
* LIFO queue(last-in,first-out,aka a stack)
* Priority(ưu tiên) queue

**Function** Breadth-first search (problem) **return** một solution hoặc failure

Ban đầu nó sẽ tạo ra cái node

Node 🡨a node with STATE problem .INITIAL-STATE,PATH -COST=0

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

Frontier 🡨 a FIFO 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.ACTION(node.STATE) **do**

Child 🡨 CHILD-NODE(problem,node,action)

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

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

Frontier 🡨 INSERT(child,fromtier)

BFS có thể tìm được tối ưu solution.

**Code by python**

|  |
| --- |
| A screenshot of a cell phone  Description automatically generated |

* + - **Uniform-cost search algorithm (UCS)**

**Function** UNIFORM-COST-SEARCH(problem) **returns** a solution, or failure

node f— a node with STATE = problem.INITIAL-STATE, PATH-COST = 0

frontier f— a priority queue ordered by PATH-COST, with node as the only element

explored f— an empty set

**Loop do**

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

node f— 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 f— CHILD-NODE(problem, node, action)

**If** child.STATE is not in explored or frontier **then**

frontier f— INSERT(child, frontier)

**Else if** child.STATE is in frontier with higher PATH-COST **then**

replace that frontier node with child

**Code by Python**

|  |
| --- |
| **A screenshot of a cell phone  Description automatically generated** |

* **So sánh những thuật toán**

So sánh những thuật toán bằng những tiêu chuẩn sau:

* Completeness: tìm được solution nếu nó tồn tại.
* Optimality: optimal solution (tối ưu)
* Complexity
  + Time
  + Space
* Thuật toán BFS sẽ có :
  + Completeness
  + Optimality
* **Depth-first search (DFS)**

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

node f— a node with STATE = problem.INITIAL-STATE, PATH-COST = 0

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

frontier f— a LIFO queue with node as the only element

explored f— an empty set

**Loop do**

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

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

add node.STATE to explored

**for each** action **in** problem.ACT10NS(node.STATE) **do**

child f— 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 f— INSERT(child,frontier).

**Code by python**

**A screenshot of a cell phone

Description automatically generated**

Đôi khi DFS có thể không tìm được solution.

* Để giải quyết vấn đề ta cần một thuật toán DEPTH-LIMITED SEARCH
* **Depth-limited search(DLS)**

Về cơ bản nó là DFS nhưng thuật toán này cho là cho phép dừng expanding khi nó đạt tới depth limit.

**function** DEPTH-LIMITED-SEARCH(problem, limit) **returns** a solution, or failure/cutoff

**return** RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE),problem,limit)

**function** RECURSIVE-DLS(node, problem, limit) **returns** a solution, or failure/cutoff

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

**else if** limit = 0 then return cutoff

**else**

cutoff-occurred? f— false

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

child f— CHILD-NODE(problem, node, action)

result 4-— REcURsIVE-DLS(child, problem, limit — 1)

**If** result = cutoff then cutoff-occurred? true

**else** if result # failure then return result

**if** cutoff-occurred? then return cutoff else return failure

**Code by python**

**A flat screen tv

Description automatically generated**

Vấn đề của depth-limited search : phải xác định được depth limit. Để giải quyết được nó ta sẽ phải có thêm thuật toán nữa : Iterative deepending search.

* **Interative deepending search**

**function** ITERATIVE-DEEPENING-SEARCH(problem) **returns** a solution, or failure

for depth = 0 to oc do

result f— DEPTH-LIMITED-SEARCH(problem, depth)

**if** result # cutoff then return result

Đây là thuật toán tốt nhất những cái ta đã biết vì có sự kết hợp giữa depth-first và breadth-first

**Code by python**

**Screen of a cell phone

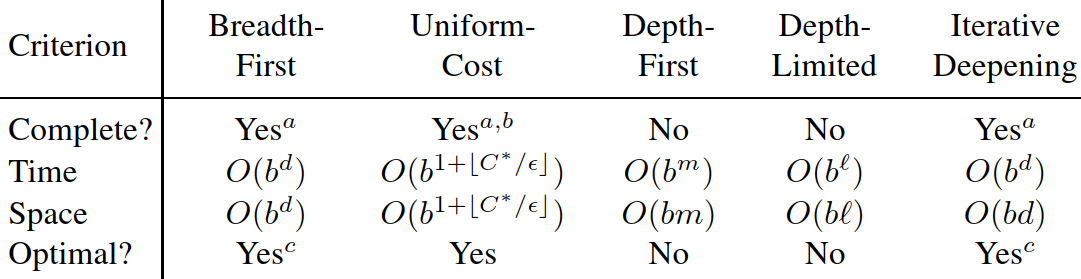
Description automatically generated**

* **So sánh giữa DFS và BFS**

BFS là optimal ( điều kiện ) còn DFS là nonoptimal.

DFS tốt hơn BFS trong space complexity.

* **So sánh những thuật toán đã học**

****

* 1. **Informed search algorithms**

Tìm kiếm có nhiều sự hổ trợ khác giúp tìm ra solution nhanh chóng hơn.

**Function** Informed-SEARCH(problem) **returns** a solution, or failure

node f— a node with STATE = problem.INITIAL-STATE, PATH-COST = 0

frontier f— a priority queue ordered by f(n), with node as the only element

explored f— an empty set

**Loop do**

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

node f— 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 f— CHILD-NODE(problem, node, action)

**If** child.STATE is not in explored or frontier **then**

frontier f— INSERT(child, frontier)

**Else if** child.STATE is in frontier with higher PATH-COST **then**

replace that frontier node with child

* + - **Evaluation function f(n)**

Hàm dùng để ước lượng các chi phí của các node ( Đây là chi phi đi tới goal)

Ta có thể dùng f(n) để mở rộng các nút, ta lựa chọn node nào được chọn để mở rộng 🡪 ưu tiên cost nào thấp hơn .

Lưu ý : các lựa chọn khác nhau của f(n) sẽ dẫn tới thuật toán khác nhau của Informed search

* + - **Best-first search**

f(n) = h(n)

Heuristic function h(n) ước tính chi phí từ node tới goal.

**Code by python**

A screenshot of a cell phone

Description automatically generated

* Tuy nhiên Best-first search không phải OPTIMAL mặc dù nó chạy nhanh hơn Dept-first search.
  + - **A\* search**

f(n) = n.PATH-COST + h(n)

A\* là sự kết hợp của **complete** và **optimal.**

Một heuristic satisfies thỏa mãn như sau:

h(n) ≤ cost(n, n’) + h(n’)cho tất cả các node và successor.

Một heuristic không thỏa mãn tính chất trên sẽ dẫn tới cost lớn hơn so với bình thường

* + - **So sánh thuật toán**

If h(n) = 0 for all n, then f(n) = n.PATH-COST

* A\* nó giống UCS về độ tối ưu

Cùng với những heuristic thống nhất 🡪 A\* có thể tìm ra optimal solution một cách rất là nhanh.

Lưu ý : A\* là một biến thể hoàn hảo hơn của Best-frist-search.

Code by python

A screenshot of a cell phone

Description automatically generated

* + - **Generating heuristic functions**
  1. **Search in more complex environments**
* **Local search**

Chỉ sử dụng duy nhất node hiện tại và những successor.

* **Hill-climbing search**

**function** HILL-CLIMBING(problem) **returns** a state that is a local maximum

current f— MAKE-NODE(problem.INITIAL-STATE)

**loop do**

neighbor f— a highest-valued successor of current

**if** neighbor. VALUE current <= VALUE **then return** current.STATE

current f— neighbor

Những vấn đề của Hill-climbing search

* Sinh ra quá nhiều successors.
* Thường hay bị mắc kẹt tại một vị trí nào đó ( local max)

Để khắc phục những vấn để trên a nên cho phép thuật toán di chuyển theo hàng ngang

**function** HILL-CLIMBING(problem) **returns** a state that is a local maximum

current f— MAKE-NODE(problem.INITIAL-STATE)

**loop do**

neighbor f— a highest-valued successor of current

**if** neighbor. VALUE current < VALUE or **sideways movers > limit** **then return** current.STATE

current f— neighbor

Để có thể tối ưu giải thật Hill-climbing search

Phương pháp 1: **Random-restart hill climbing**:

Phương pháp 2: **Local beam search**

Thay vì thử k tìm kiếm ngẫu nhiên liên tục, hãy thực hiện chúng đồng thời

* Nhanh hơn rất nhiều

Có một dạng cái tiếng của Local beam search để tang hiệu xuất đó là kết hợp với Stochastic hill climbing.

Phương pháp 3: **Simulated annealing**

Giúp đạt cost thấp nhất.

* **Simulated annealing algorithm**

**function** SIMULATED-ANNEALING(problem, schedule) **returns** a solution state

**inputs**: problem, a problem

schedule, a mapping from time to "temperature"

current f— MAKE-NODE(problem.INITIAL-STATE)

for t = 1 to oc do

T schedule(t)

**if** T = 0 then return current

next f— a randomly selected successor of current

AE f— next. VALUE — current. VALUE

**if** AE > 0 then current f— next

else current f— next only with probability eAE/T

* 1. **Searching in NONDETERMINISTIC ENVIRONMENTS**

Có thể hiểu đơn giản như : 1 action 🡪 nhiều successor .

* **The erratic vacuum world**

Điển hình như máy hút bụi hư , 1 action của nó có thể hút nhưng lại vừa có thể thả bụi ra.

Để giải quyết bài toán ta cần thay đổi 2 thứ:

**Transition model** : RESULT(s,a) = {s1,s2,…}.

**Solution :** conditional plan 🡪 kế hoạch có điều kiện ( tùy theo mỗi trạng thái của môi trường mà hành động khác nhau.

* **AND – OR search tree**

Giống search tree bài trước nhưng khác là sẽ có them 2 loại node:

OR nodes: states

AND nodes: khi ra kết quả nó có thể ra nhiều node

* **AND – OR graph search algorithm**

**function** AND-OR-GRAPH-SEARCH(probIem) **returns** a conditional plan, or failure

OR-SEARCH(problem.INITIAL-STATE, problem, [

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

**if** problem.GOAL-TEST(state) **then return** the empty plan

**if** state is on path **then return** failure

**for each** action in problem.ACTIONS(state) **do**

plan e— AND-SEARCH(RESULTS(state, action), problem, [state I path])

**if** plan # failure then return [action I plan]

**return** failure

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

**for each** Si in states **do**

plani e— OR-SEARCH(si, problem, path)

**if** plani = failure then return failure

**return** [if Sl then planl else if s2 then plan2 else

. **if** sn—l then plann\_l else plann

* **Partially observable environments**

Giống như nondetermimistic environments nhưng agents nó không có đủ percept, nó không biết vị trí state của mình

* **Sensorless problems**

Trường hợp hoàn toàn không có observation, bất kì thông tin nào về môi trường

* Giải quyết rất ít vấn đề
* Tiết kiệm được một vài chi phí

Belief state : Agent nghĩ nó đang ở đâu đó 🡪 là một tập hợp state.

* Ý tưởng để giải quyết bài toán này là xây dựng hệ thống có con agent này có chức năng belief state 🡪 để nó trở thành fullobservable

Gồm 5 thành phần:

* Initial sate

Lấy tấ cả physical states

* Possible actions

ACTION(belief state b)

* Transition model

RESULT( belief state b,action a) = new belief state b’

* Goad test : Result
* Cost
* Qua các bước sau 🡪 ta có thể sử dụng những thuật toán trước đã học để giải quyết bài toán

Tuy nhiên, mô hình state giờ rộng lớn hơn nhiều. Máy sẽ không thế chạy nổi.

* Ta cần cái tiển thuật toán 🡪 incremental belief-state search
* **Incremental belief-state search**

Solve 1 physical state tại một thời điểm

* 1. **Searching with no transition model – online search**
* **Offline và online**

Tất cả thuật toán trước giờ là offline

Gồm 5 thành phần:

* Initial state
* Possible actions:ACTIONS
* Transition model: RESULT (online không có thành phần này)
* Goal test: GOAL-TEST
* Cost: STEP-COST,PATH-COST

|  |  |
| --- | --- |
| Offline Search | Online Search |
| Agent biết trước môi trường sẽ như thế nào.  Mô phỏng trong máy tính  Solotion 🡪 agents | Đây con agent vào môi trường  Agent giải quyết solution |

* **Lợi ích ủa Online search**
* Đây là hình phạt cho quá nhiều compitation
* Không cần điều kiện từ môi trường
* Không environment model
* **Competitive ratio**

Có 2 loại path cost trong online search

* Actual path cost
* Shortest path cost

Competitive raio = Actual path cost/ Shortest path cost

* **Dead-en state**

Một vài actions là irreversible (đảo ngược) , ta sử dụng dead-end state

* **Online depth-first search agent**

**function** ONLINE-DFS-AGENT(s') **returns** an action

**inputs**: s', a percept that identifies the current state

**persistent**: result, a table indexed by state and action, initially empty

untried, a table that lists, for each state, the actions not yet tried

unbacktracked, a table that lists, for each state, the backtracks not yet tried

s, a, the previous state and action, initially null

**if** GOAL-TEST(s') **then return** stop

**if** s' is a new state (not in untried) then untried[s'] 4— ACTIONS(s')

**if** s is not null then

result[s, a] s'

add s to the front of unbacktracked[s']

**if** untried[s'] is empty then

**if** unbacktracked[s'] is empty **then return** stop

else a an action b such that result[s', b] =

else a c—

**return a**

* **Tại sao lại sử dụng depth-first?**

Trong online search: những agent actually walks ( đi bất cứ đâu) trong environment

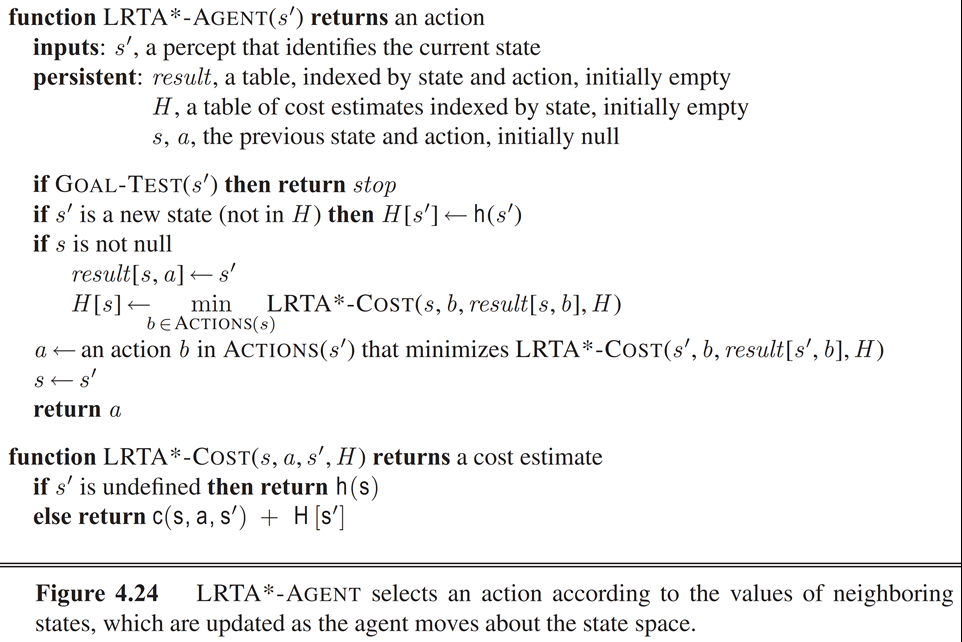
In depth-first search: những agent chỉ có di chuyển điểm tiếp theo ( cùng 1 hoặc 1 vài actions)

Tuy nhiên BFS 🡪 không tối ưu

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

Ý tưởng :

Mỗi bước các agent di chuyển các successor sao cho ít tổn phí nhất



* **Constraint satisfaction problems(CSP)**
* **Solve CSPs using Constrain Propagation.**

Ta vẫn có thể giải quyết vấn sử dụng searching on state space tuy nhưng không nhanh bằng CSP.

* **Constraint propagation**

Sử dụng các ràng buộc để giảm tên miền (giá trị pháp lý) cho một biến

* **Local consisitency**

**Enforcing local consistency** trong mỗi phần trong graph 🡪 inconsistent

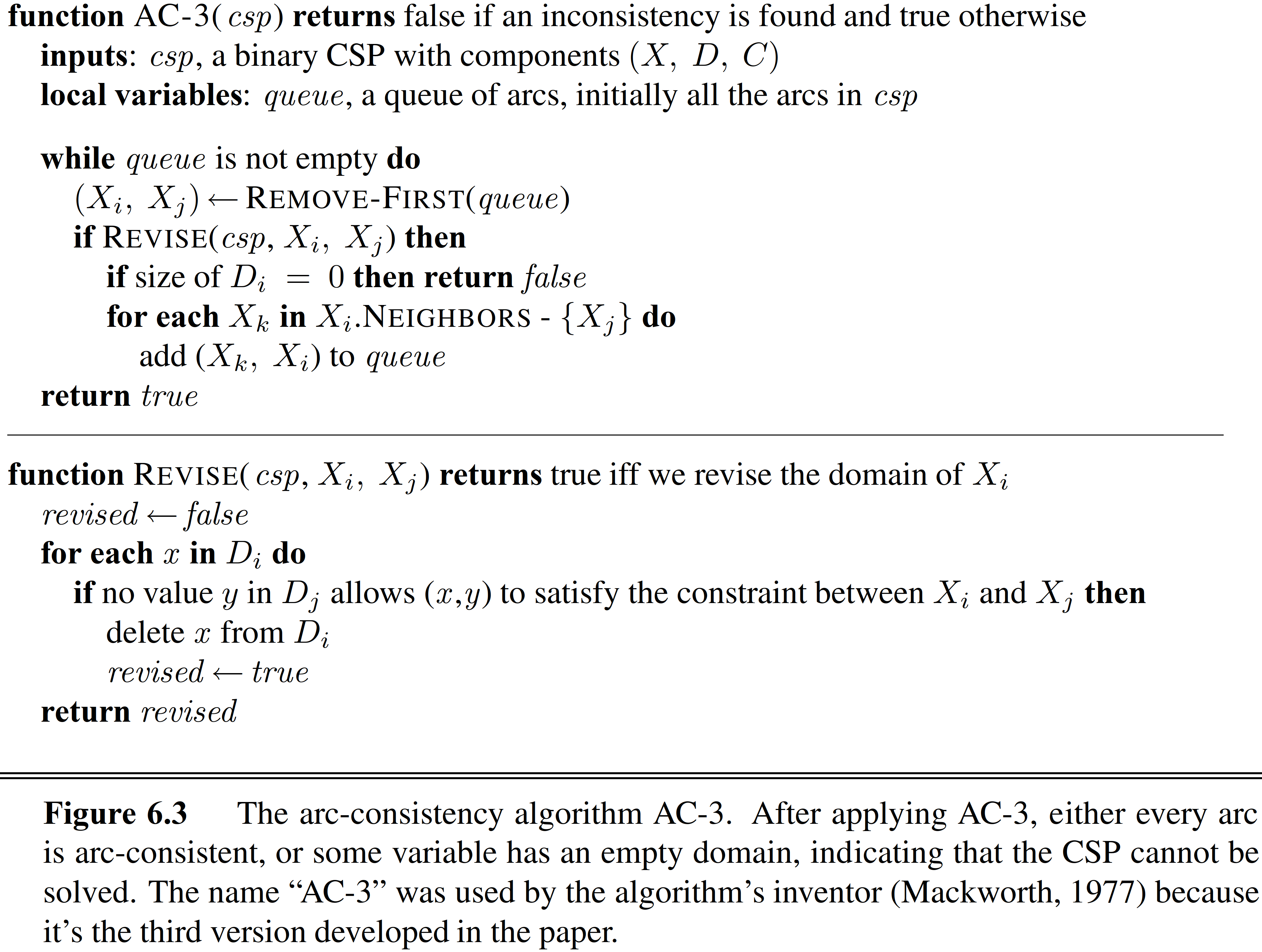
* **Node consistency**

Thực thi các tên miền để đáp ứng các ràng buộc đơn nhất.

* **Arc consistency**

Thực thi domains sao cho satisfy rang buộc nhị phân

* **AC-3 algorithm**

****

* **Path consistency**

Arc có thể giúp giảm tên miền tuy nhiên nó lại bị no effect.

* **K-consistency**

Là một khung chung của local consistency.

1 CSP là một k-consistency.

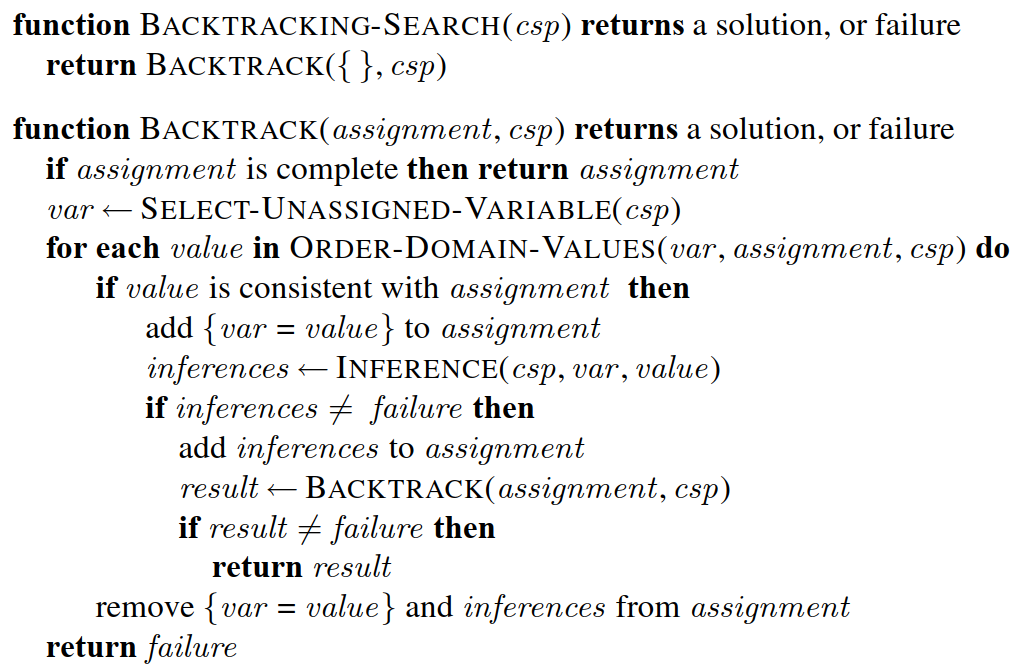
* **Global constraints**

Một global constraints sẽ giống như một arbitrary number of variables.

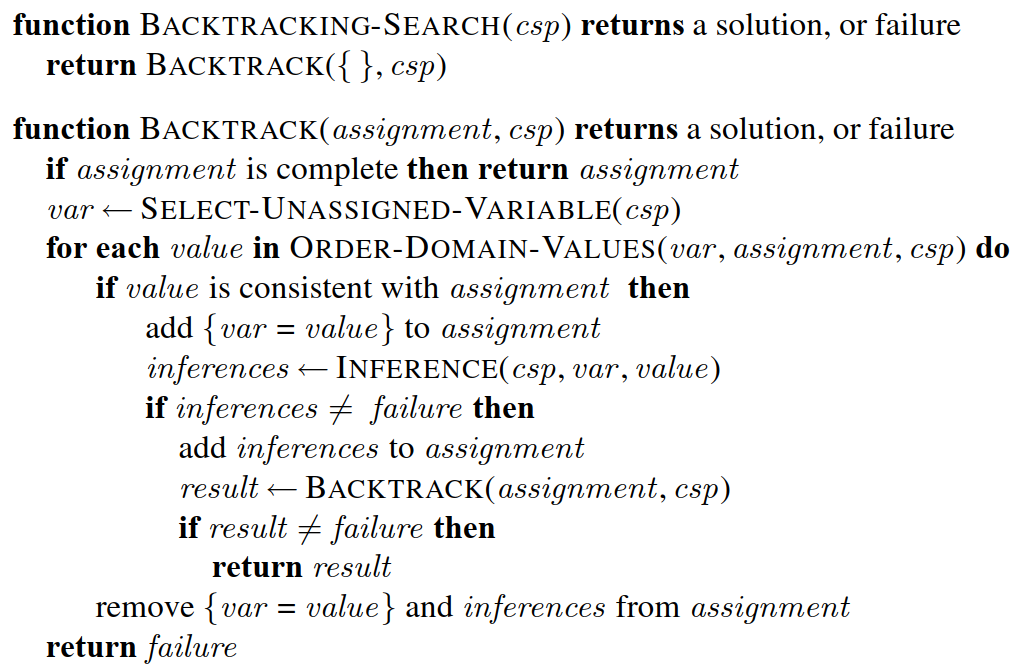
* **Solve CSPs using Backtracking Search**

Ý tưởng: một depth first search chọn ra biến duy nhất tại một thời điểm nhất định và backtrack khi 1 biến no legal values left to assign.

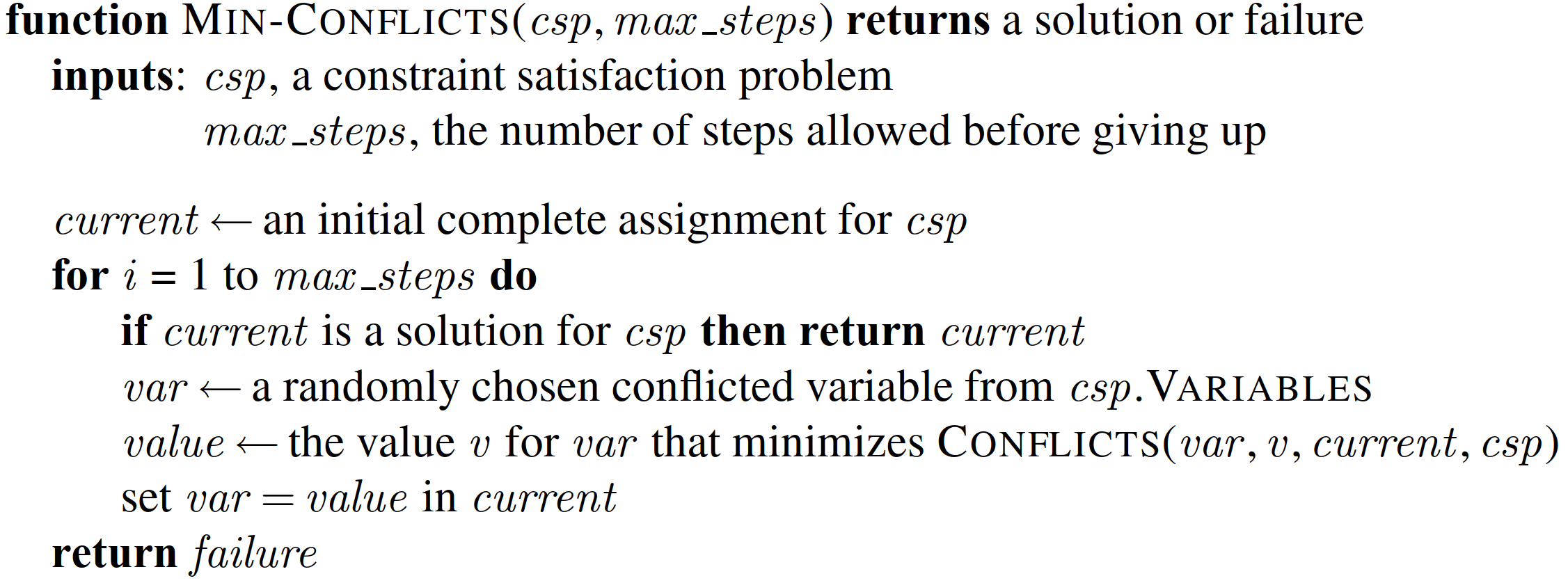
Backtracking search uses **CSP compinents.**

****

* **Comments on backtracking search algorithm**

****

* **Min-conflicts algorithm**

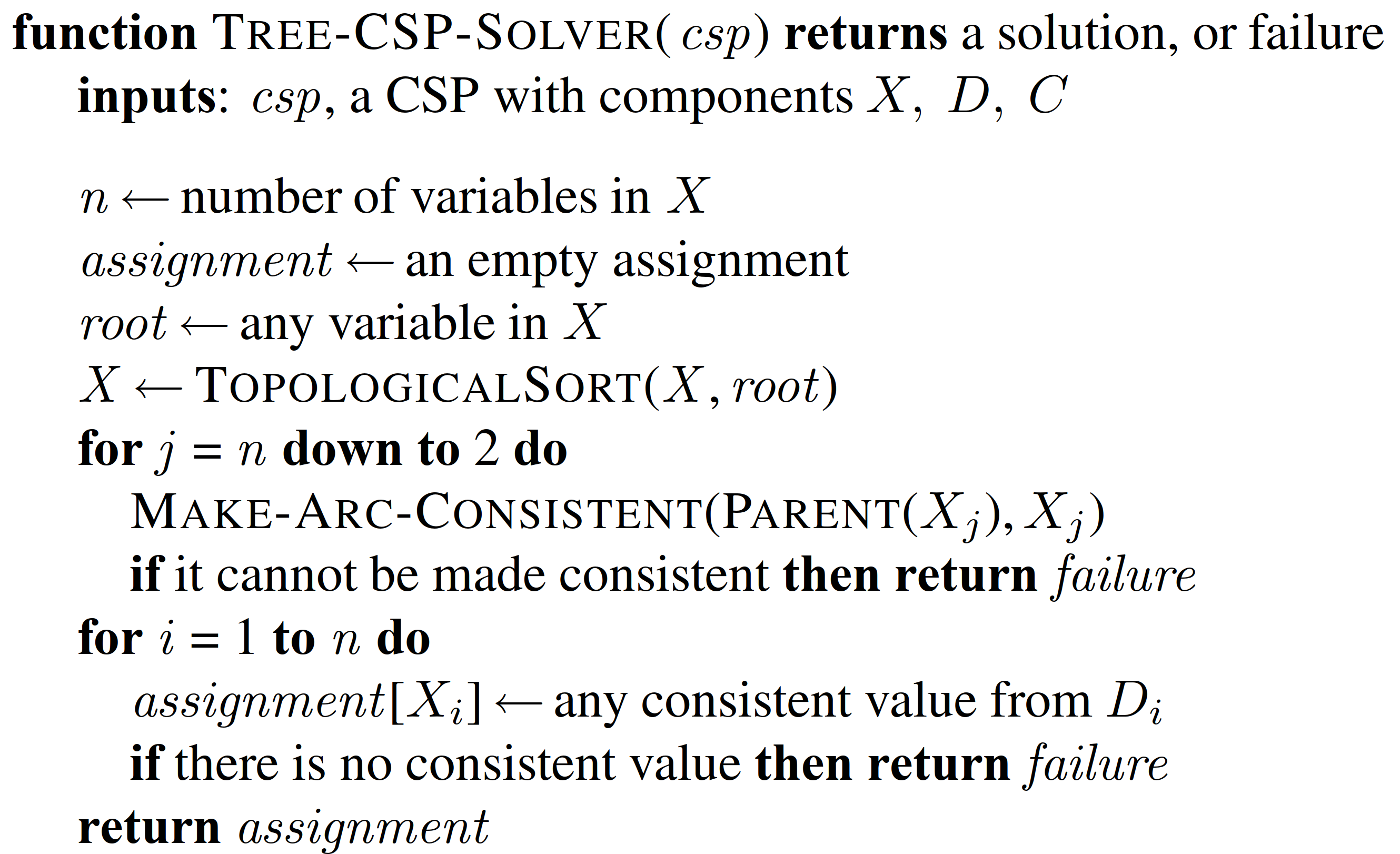


Min-confly is hiệu quả hơn rất nhiều so với CSPs

Nó có thể giải quyết số lượng problem rất lớn.

Thời gian giái quyết lại rất nhanh.

* **Constraint graph**
* **Tree CSP solver**

****

* **Reinforcement learning**
* **Cross – entropy method**

1. **Tài liệu tham thảo**

* **Những video giảng của thầy Trần Nhật Quang**
* [**https://chame.rmit.edu.vn/dao-van-la-gi-va-lam-sao-de-giup-con-khong-dao-van/**](https://chame.rmit.edu.vn/dao-van-la-gi-va-lam-sao-de-giup-con-khong-dao-van/)
* [**https://vdodata.vn/tri-tue-nhan-tao-la-gi-lich-su-phat-trien-tri-tue-nhan-tao-ai/**](https://vdodata.vn/tri-tue-nhan-tao-la-gi-lich-su-phat-trien-tri-tue-nhan-tao-ai/)
* [**https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/**](https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/)
* **Tham thảo code tại**

**https://github.com/tiendung1510/LearningPython**