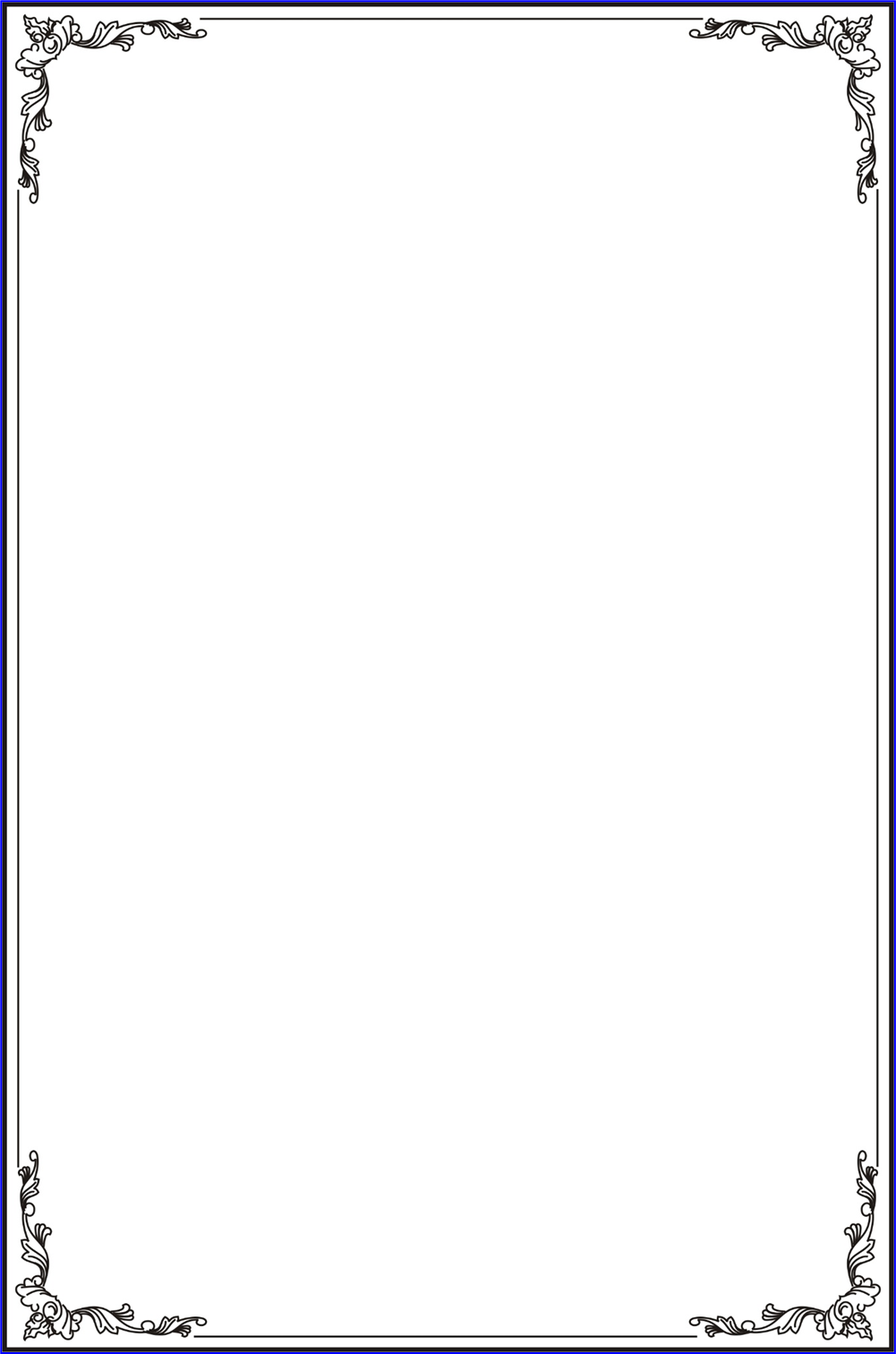
****

**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

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

**KHOA CÔNG NGHỆ THÔNG TIN**

**🕯✡🕮🕮✡🕯**



**LỜI NÓI ĐẦU**

**Báo cáo:**

**TRÍ TUỆ NHÂN TẠO**

|  |  |
| --- | --- |
| ***Nhóm sinh viên thực hiện :*** | |
| Hồ Huy Hoàng | 18110284 |
| Tạ Thị Mai Hương | 18110298 |
| Lâm Quốc Khánh | 18110304 |
| Trần Quốc Nam | 18110323 |
| Nguyễn Trần Phúc | 18110339 |
| Nguyễn Huỳnh Minh Tiến | 18110377 |

**Giảng viên : Trần Nhật Quang**

**TP. Hồ Chí Minh, tháng 6 năm 2020**

Trí tuệ nhân tạo đã xuất hiện được một thời gian dài, nhưng đến hiện tại thì vẫn chưa được phổ biến và xã hội hóa. Vậy nên dù có khá nhiều thông tin nhưng tìm hiểu về AI vẫn không phải là dễ. Qua bản báo cáo này, nếu có gì sai sót mong thầy thông cảm và góp ý để chúng em hoàn thiện hơn.

Để hoàn thành được báo cáo về môn học Trí tuệ nhân tạo, chúng em không thể quên gửi lời cảm ơn đến Giảng viên. Thầy Trần Nhật Quang đã cung cấp nguồn kiến thức và tài liệu để chúng em học tập, tìm hiểu.

Chúng em xin chân thành cảm ơn!

Tập thể nhóm

|  |  |
| --- | --- |
| **Điểm** | **Đánh giá của giảng viên** |
|  |  |

MỤC LỤC

[I. ĐẠO VĂN 7](#_Toc43677680)

[II. TÌM HIỂU VỀ TRÍ TUỆ NHÂN TẠO (ARTIFICIAL INTELLIGENCE) 8](#_Toc43677684)

[1. Trí tuệ nhân tạo là gì ? 8](#_Toc43677685)

[2. Lịch sử hình thành AI 8](#_Toc43677686)

[3. Ứng dụng của Trí tuệ nhân tạo 9](#_Toc43677690)

[III. ĐẠO ĐỨC TRÍ TUỆ NHÂN TẠO 10](#_Toc43677691)

[IV. THUẬT TOÁN TÌM KIẾM 11](#_Toc43677693)

[1. Các bước cơ bản của một thuật toán tìm kiếm 11](#_Toc43677694)

[2. Tiêu chí đánh giá thuật toán 11](#_Toc43677695)

[3. Conceptual algorithm 12](#_Toc43677696)

[4. Hai loại thuật toán tìm kiếm 13](#_Toc43677699)

[5. Các cấu trúc dữ liệu 13](#_Toc43677700)

[6. Các thuật toán Uninformed search 14](#_Toc43677703)

[6.1. Thuật toán Breadth-ﬁrst search (BFS) 14](#_Toc43677704)

[6.2. Thuật toán Uniform-cost search 14](#_Toc43677705)

[6.3. Thuật toán Depth-ﬁrst search (DFS) 15](#_Toc43677706)

[6.4. Thuật toán Depth-limited search 15](#_Toc43677707)

[6.5. Thuật toán Iterative deepening search 16](#_Toc43677708)

[7. Các thuật toán Informed search 17](#_Toc43677709)

[7. 1. Các hàm ước lượng quảng đường 17](#_Toc43677710)

[7.2. Ý tưởng chung của Informed search 17](#_Toc43677711)

[7.3. Thuật toán Best-first search 17](#_Toc43677712)

[7.4. Thuật toán A\* search 17](#_Toc43677713)

[7.5. Cách tạo heuristic function 18](#_Toc43677714)

[V. LOCAL SEARCH – SEARCHING FOR GOAL STATE 19](#_Toc43677715)

[1. So sánh với các thuật toán tìm kiếm trước đó: 19](#_Toc43677716)

[2. State space landscape: 19](#_Toc43677717)

[3. Local search: 20](#_Toc43677718)

[4. Hill-climbing search: 20](#_Toc43677724)

[5. Issues of Hill-climbing search: 21](#_Toc43677730)

[5.1. Quá nhiều successors: 21](#_Toc43677731)

[5.2. Local optimum: 22](#_Toc43677732)

[6. Simulated Annealing: 24](#_Toc43677733)

[7. Searching in NONDETERMINISTIC environments 26](#_Toc43677740)

[7.1. Tìm kiếm trong môi trường không xác định 26](#_Toc43677741)

[7.2. And\_Or Graph Search 27](#_Toc43677742)

[8. Searching in Partially Observable Enviroments 30](#_Toc43677743)

[8.1. No observable 30](#_Toc43677744)

[8.2. Partially observable 30](#_Toc43677745)

[8.3. Demo 31](#_Toc43677746)

[9. Online Search 32](#_Toc43677747)

[9.1. So sánh về offline search và online search : 32](#_Toc43677748)

[9.2. Competitive ratio : 32](#_Toc43677749)

[9.3. Online Depth-first search agent : 32](#_Toc43677750)

[9.4. Online A\* search : ( tìm được solution tốt một cách nhanh chóng ) 34](#_Toc43677751)

[9.5. Learning Real – Time A\* agent : 34](#_Toc43677752)

[10. Constraint Satisfactoin Problems 35](#_Toc43677753)

[10.1. Constraint satisfaction problems (CSP) 35](#_Toc43677754)

[10.2. Constraint graph : 35](#_Toc43677755)

[10.3. Constraint propagation : 36](#_Toc43677756)

[10.4. Local consistency (node , arc , path ) 36](#_Toc43677757)

[10.5. AC – 3 algorithm (arc consistency) 36](#_Toc43677758)

[10.6. Path consistency : 38](#_Toc43677759)

[10.7. K-consistency : (dạng tổng quát của local consistency) 38](#_Toc43677760)

[10.8. Global constraints : 38](#_Toc43677761)

[10.9. Solving a Sodoku puzzle : (AC – 3, Triplets ) 38](#_Toc43677762)

[10.10. Backtracking search algorithm 38](#_Toc43677763)

[10.10. Comments on backtracking search algorithm : 40](#_Toc43677764)

[TÀI LIỆU THAM KHẢO 41](#_Toc43677765)

I. ĐẠO VĂN

# Đạo văn là gì?

* Đạo văn là hành vi thiếu trung thực về mặt học thuật, là hành vi vi phạm đạo đức nghiêm trọng và đôi khi có thể cấu thành hành vi vi phạm bản quyền.
* Những biểu hiện của đạo văn:
  + Sao chép ý tưởng, ngôn từ của người khác thành của mình.
  + Sử dụng hoàn toàn ý tưởng và văn phong của người khác mà không ghi nguồn.
  + Đôi khi, dù có dẫn nguồn nhưng việc sao chép quá nhiều cũng có thể xem là đạo văn.

# Những điều nên/không nên làm khi làm báo cáo:

* Không sử dụng ngôn từ, code của người khác mà không ghi nguồn.
* Không sao chép hoàn toàn ý tưởng, sao chép code của người khác.
* Tự tìm hiểu và hoàn thành báo cáo theo cách hiểu của mình.
* Chuẩn bị đầy đủ nội dung kiến thức về lý thuyết và thực hành khi báo cáo.

# Cam kết:

Chúng em xin cam đoan đồ án này do các thành viên nhóm thực hiện. Nếu phạm lỗi đạo văn (như: sử dụng tài liệu, code của người khác mà không ghi rõ nguồn, sao chép trên 30% báo cáo), chúng em xin chịu hoàn toàn trách nhiệm.

|  |  |
| --- | --- |
| **Họ tên** | **MSSV** |
| Hồ Huy Hoàng | 18110284 |
| Tạ Thị Mai Hương | 18110298 |
| Lâm Quốc Khánh | 18110304 |
| Trần Quốc Nam | 18110323 |
| Nguyễn Trần Phúc | 18110339 |
| Nguyễn Huỳnh Minh Tiến | 18110377 |

II. TÌM HIỂU VỀ TRÍ TUỆ NHÂN TẠO (ARTIFICIAL INTELLIGENCE)

# 1. Trí tuệ nhân tạo là gì ?

* Trí tuệ là gì
* Nhân tạo là những sản phẩm do con người tạo ra, không có từ tự nhiên

# 2. Lịch sử hình thành AI

## 2.1. Vở kịch Rossumovi Univerzální Roboti

* Năm 1920, một nhà văn học người Séc có tên Karel Čapek cho ra đời vở kịch khoa học viễn tưởng tên Rossumovi Univerzální Roboti (Rossum’s Universal Robots, viết tắt là R.U.R). Đây là vở kịch thú vị và nó được xem như cột mốc đầu tiên trong lịch sử hình thành AI.
* Nội dung vở kịch kể về một nhà máy sử dụng các chất hữu cơ để chế tạo ra con người nhân tạo gọi là robot. Các robot này khác với định nghĩa robot ngày này, nó được tạo ra bằng thịt và máu chứ không phải máy móc. Lúc đầu, robot là một công cụ giúp ích cho con người rất nhiều trong công việc. Nhưng sau đó chúng nổi loạn và dẫn đến sự tuyệt chủng của con người.
* Mặc dù ý tưởng robot trong vở kịch không phải là ý tưởng của robot ngày này như nó cũng đã giới thiệu thuật ngữ robot và một loại trí tuệ nào đó do con người tạo ra vừa có ích cho con người, đồng thời cũng mối đe dọa cho con người nếu không kiểm soát được chúng.
* Trí tuệ nhân tạo trong nghệ thuât là một đề tài rất khác nhưng nó cũng ảnh hưởng phần nào đến việc nghiên cứu Trí tuệ nhân tạo trong khoa học và ứng dụng trong cuộc sống.

## 2.2. Alan Turing và phép thử



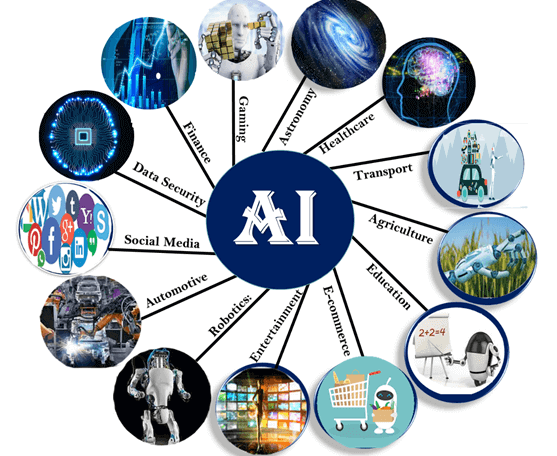
**Alan Turing (1912–1954)**

* Alan Turing là một nhà toán học, logic học và mật mã học người Anh, ông được xem là cha đẻ của ngành Khoa học máy tính nói chung và Trí tuệ nhân tạo nói riêng.
* Vài năm sau chiến tranh thế giới thứ hai, Alan Turing giới một phép thử (phép thử Turing). Phép thử được trong bài viết "Computing Machinery and Intelligence" khi ông đang công tác tại trường Đại học Manchester vào năm 1950.
* Phép thử Turing dùng để xác định trí thông minh của máy. Theo phép thử, một máy tính có thể được coi là thông minh nếu máy tính (A) và một người (B) cùng giao tiếp bằng ngôn ngữ tự nhiên với một người khác (C), mà người này (C) không thể phân biệt được đâu là máy (A hay B)

## 2.3. Thuật ngữ "artificial intelligence"và hội nghị Dartmouth (1955–1956)

* Cùng với Alan Turing, Marvin Minsky, Allen Newell, Herbert A. Simon, **John McCarthy** là một trong những người cha sáng lập nên Trí tuệ nhân tạo. McCarthy đặt ra thuật ngữ "artificial intelligence" vào năm 1955, đó cũng chính là chủ đề của hội nghị Dartmouth vào mùa hè năm 1956 do ông tổ chức.
* Hội nghị Dartmouth có lẽ là workshop đầu tiên về Trí tuệ nhân tạo và hình thành nên khoa học nghiên cứu về Trí tuệ nhân tạo.
* Các nhà nghiên cứu từ Đại học Carnegie Mellon, Học viện Công nghệ Massachusetts, các nhân viên từ công ty IBM đã gặp nhau tại hội nghị và cùng nhau nghiên cứu về thành lập các phòng nghiên cứu Trí tuệ nhân tạo.
* Một vài năm sau đó, những nghiên cứu về Trí tuệ nhân tạo đạt được thành công vượt bậc. Vào thời điểm này, tất cả các chuyên gia về Trí tuệ nhân tạo đều vô cùng lạc quan về sự phát triển trong tương lai của công nghệ này.
* Sau đó Trí tuệ nhân tạo cũng có những giai đoạn chữg lại như những năm 1970, 1987-1997.
* Thế kỷ 21 hứa hẹn là thời gian phát triển mạnh mẽ nhất của Trí tuệ nhân tạo với những ứng dụng như Big Data, Deep Learning và ứng dụng thương mại rộng rãi …

# 3. Ứng dụng của Trí tuệ nhân tạo

Trí tuệ nhân tạo được ứng dụng trong nhiều lĩnh vực thực tế

Nguồn ảnh : Application of AI, https://www.javatpoint.com/application-of-ai

III. ĐẠO ĐỨC TRÍ TUỆ NHÂN TẠO

Thời đại phát triển, công nghệ đang và dần thay thế sức người trong mọi việc. Không thể không kể đến là các ứng dụng của AI vào những việc mà đôi khi ta không thể hình dung. Sự bùng nổ của AI sẽ đem đến rất nhiều những lợi ích cho con người, bên cạnh đó cũng có những mặt xấu.

Vậy AI có đạo đức hay không?

# Các trường phái đạo đức của AI:

**Virtue ethics:**

Trường phái tuân theo đức hạnh. Đức hạnh là cái trọng yếu nhất của con người, là khởi nguồn cho những đức tính khác. Đã là người thì phải tu dưỡng đức hạnh đầu tiên.

**Deontology:**

Trường phái tuân theo quy luật. Đối với trường phái này, con người được xem là tốt nếu như họ hành động theo luật, và tôn trọng những điều luật đó.

**Consequentialism:**

Trường phái theo kết quả. Những hành động được xem là tốt hay xấu sẽ đều dựa vào kết quả mà nó mang lại. Nếu tốt thì hành động đó là đúng, còn không thì sẽ bị bài trừ.

Từ đây, ta có thể thấy được rằng, AI là một sản phẩm do con người tạo ra, vậy nên nó sẽ ảnh hưởng một phần từ đạo đức con người. Nhiều nhà khoa học vẫn luôn băn khoăn về việc áp dụng đạo đức lên các công cụ Trí tuệ nhân tạo. Nhưng nếu không có, đó có thể là một mối họa ngầm cho xã hội và cũng một phần làm chậm đi sự phát triển của nền công nghiệp Trí tuệ nhân tạo.

IV. THUẬT TOÁN TÌM KIẾM

# 1. Các bước cơ bản của một thuật toán tìm kiếm

* Bước 1 : Tùy theo search strategy(chiến lược tìm kiếm), chọn một node để expand
* Bước 2 : Expanding node vừa chọn

Lặp lại hai bước trên cho đến khi tìm được solution hoặc không còn node để expand . Tất cả các thuật toán đều bao gồm hai bước bước trên, chỉ khác nhau ở cách chọn node ở bước 1

# 2. Tiêu chí đánh giá thuật toán

Trước khi chúng ta đi vào thiết kế các thuật toán tìm kiếm cụ thể, chúng ta cần xem xét các tiêu chí có thể được sử dụng để lựa chọn một thuật toán trong số nhiều thuật toán khác nhau. Chúng ta có thể đánh giá hiệu suất của thuật toán theo bốn tiêu chí:

* Completeness: Thuật toán có đảm bảo tìm ra được solution (giải pháp) nếu có cho vấn đề được đặt ra hay không ?
* Optimality: Solution tìm được có phải là solution tối ưu nhất hay không (optimal solution)
* Time complexity: Thuật toán mất bao lâu để tìm được solution?
* Space complexity: Cần bao nhiêu bộ nhớ trong quá trình tìm kiếm?

Time complexity và space complexity luôn được xem xét đối với một số biện pháp của vấn đề khác nhau. Trong khoa học máy tính lý thuyết, thước đo điển hình là kích thước của đồ thị không gian trạng thái, | V | + | E |, trong đó V là tập hợp các đỉnh (vertices - nodes) của đồ thị và E là tập hợp các cạnh (edges, links). Điều này hoàn toàn hợp lý khi đầu vào là một đồ thị có cấu trúc dữ liệu rõ ràng (Ví dụ như : bản đồ đường đi giữa các tỉnh/thành phố)

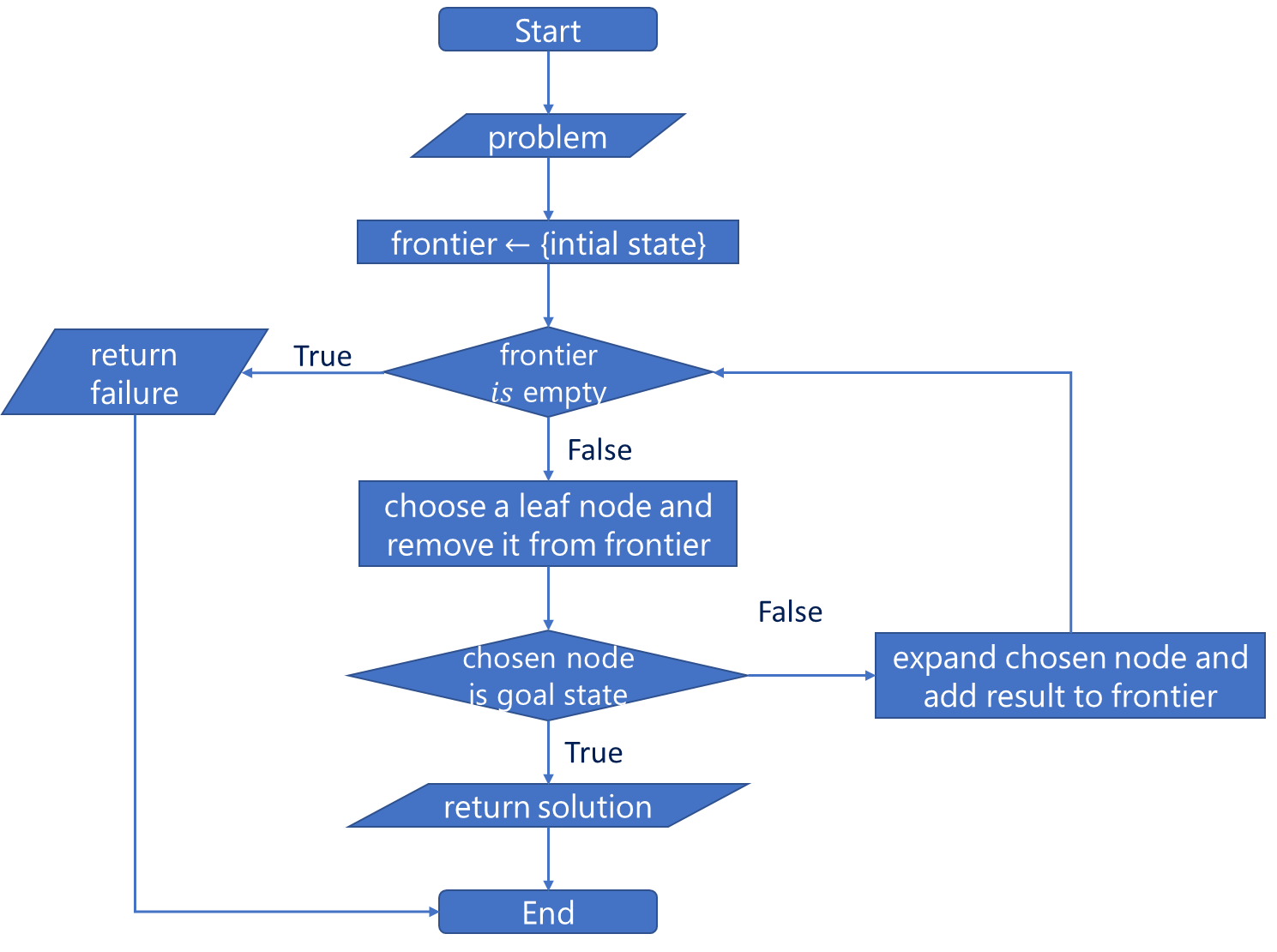
Trong Trí tuệ nhân tạo, đồ thị thường được biểu thị ngầm định bởi trạng thái ban đầu (initial state), hành động (action) và transition model. Vì thế, độ phức tạp(complexity) được biểu thị theo ba đại lượng:

* b, hệ số phân nhánh hoặc số lượng successor (nút con) tối đa của bất kỳ nút nào
* d, độ sâu (số bước dọc theo đường đi từ gốc) của goal state nông nhất
* m, độ dài tối đa của bất kỳ đường đi nào trong state space

Thời gian thường được đo bằng số lượng nút được tạo trong quá trình tìm kiếm và space tính theo số lượng nút tối đa được lưu trữ trong bộ nhớ.

# 3. Conceptual algorithm

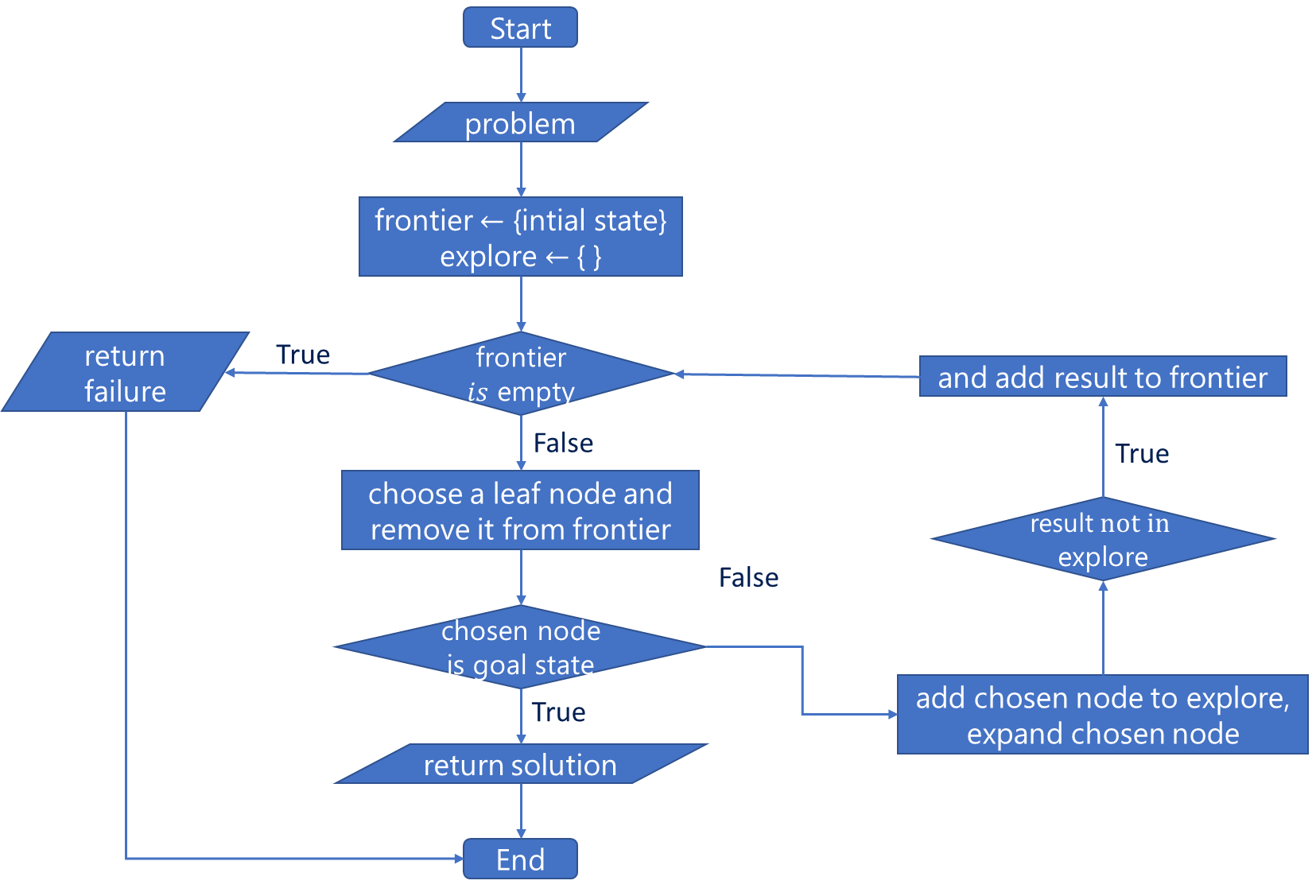
## 3.1. Conceptual algorithm #1 Tree search (cây tìm kiếm)



* Là mô hình tổng quát để xây dựng các thuật toán tìm kiếm khác
* Đầu vào : problem , đầu ra : solution
* Vấn đề : expand lại các node đã expand trước đó dẫn tới vòng lặp vô tận

## 3.2. Conceptual algorithm #2 Graph search (search with memory)

Tương tự như tree seach nhưng thêm một tập explored set chứa các node đã được expanding, tránh được vòng lặp vô tận



# 4. Hai loại thuật toán tìm kiếm

* Uninformed search (tìm kiếm mù)
  + Chỉ biết thông tin về vấn đề cần giải quyết thôi.
  + Chọn node một cách ngẫu nhiên để expand.
  + VD: Trong ván cờ, chỉ biết vị trí bắt đầu, không biết đối thủ là ai, điểm mạnh, điểm yếu của đối thủ.
* Informed search (tìm kiếm có thông tin) dựa vào thông tin có được, phỏng đoán f(n) để chọn node có giá trị f(n) nhỏ nhất để expand.
  + Ngoài thông tin về vấn đề cần giải quyết, còn có một số thông tin khác hỗ trợ giải quyết vấn đề.
  + Dựa vào thông tin có được, phỏng đoán f(n) và chọn node có giá trị f(n) nhỏ nhất để expand.
  + VD: Ngoài các vấn đề trên còn có thông tin về tìm kiếm biết con xe đi thế nào, con hậu đi thế nào

# 5. Các cấu trúc dữ liệu

## Thành phần của một Node

* State
* Parent
* Action
* Path cost

## Queue

* Toán tử của queue
  + Empty ? (queue)
  + Pop (queue)
  + Insert (element, queue)
* Các loại queue
  + FIFO Queue
  + LIFO Queue (stack)
  + Priority queue

# 6. Các thuật toán Uninformed search

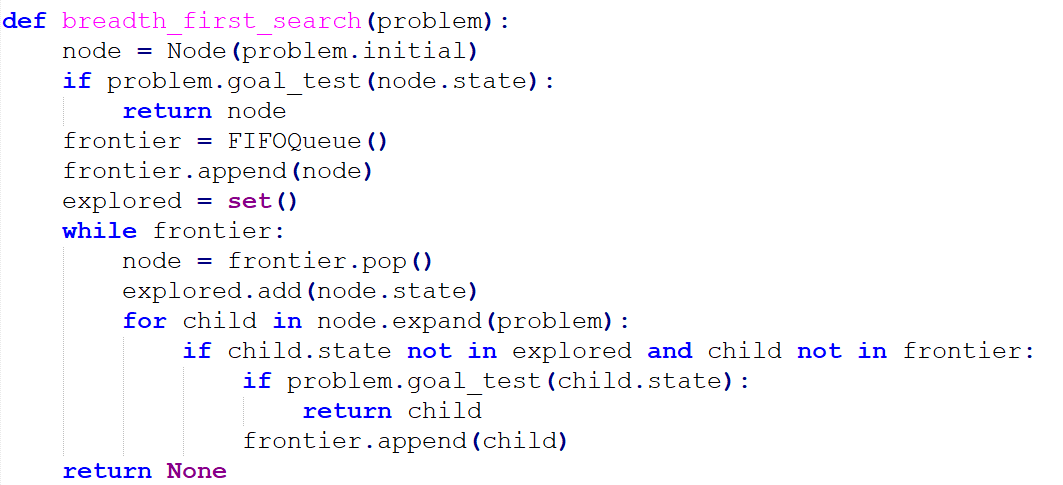
## 6.1. Thuật toán Breadth-ﬁrst search (BFS)

***Ý tưởng :***

* Là một thể hiện (instance) của Graph search
* Root node (Initial-state) được expand đầu tiên, được các successors của root, rồi tiếp expand các successors của các successors cho đến khi tìm được solution

***Đầu vào :*** problem, đầu ra : solution

***Mô tả thuật toán***



***Completeness:*** Yes

***Optimality:*** Chỉ tìm ra optimal solution nếu path cost tỉ lệ thuận với độ sâu

***Time complexity:*** Tốn thời gian

***Space complexity:*** Tốn bộ nhớ

## 6.2. Thuật toán Uniform-cost search

***Ý tưởng :*** Dựa trên BFS, thay đổi cách chọn node để expand, node có path cost nhỏ nhất được chọn trước thay vì chọn node có độ sâu nhỏ nhất

***Đầu vào:*** problem, đầu ra : solution

***Mô tả thuật toán:*** Tương tự BFS, thay đổi fronitier từ FIFO queue thành priority queue

***Completeness:*** Yes

***Optimality:*** Yes

***Time complexity:***

***Space complexity:***

Với C\* là path cost của optimal solution và step cost nhỏ nhất là

## 6.3. Thuật toán Depth-ﬁrst search (DFS)

***Ý tưởng:***

* Là một thể hiện (instance) của Graph search
* Expand những node có độ sâu lớn nhất trong frontier

***Đầu vào:*** problem, đầu ra : solution

***Mô tả thuật toán:*** Tương tự BFS, thay đổi fronitier từ FIFO queue thành LIFO queue (stack)

***Completeness:*** No

***Optimality:*** No

***Time complexity:*** , tốt hơn BFS nếu m < d

Số lượng nút trong trường hợp xấu nhất , vì vậy time complexity là

***Space complexity:*** , Ít tốn bộ nhớ hơn BFS

Mỗi nút cần lưu các nút cùng cấp và các successors. Có tối đa m nút theo chiều sâu, mỗi nút cần lưu b nút liên quan, nên space complexity là

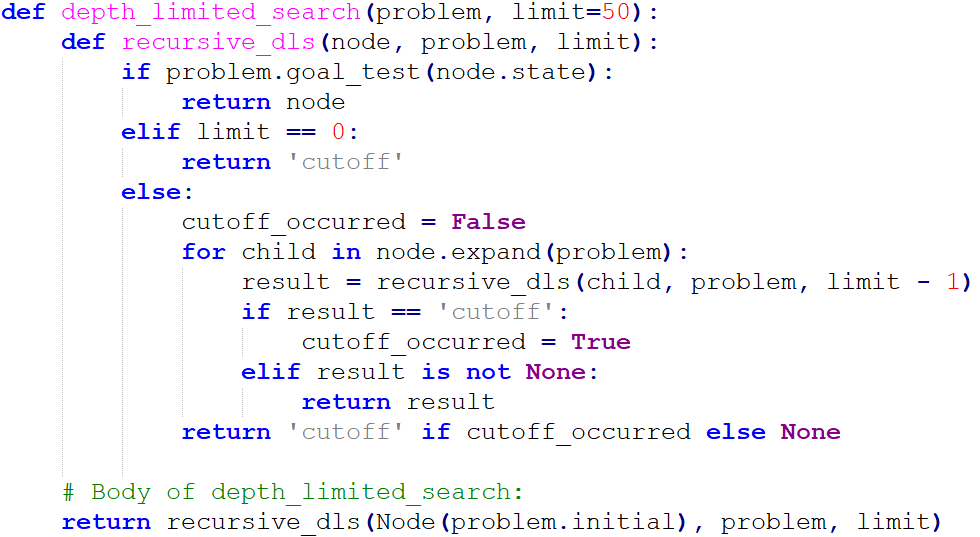
## 6.4. Thuật toán Depth-limited search

***Ý tưởng :***

Dựa trên DFS nhưng giới hạn độ sâu, nếu node chọn có độ sâu bằng giới hạn thì quay lại chọn node khác để expand

***Đầu vào :*** problem, đầu ra : solution

***Mô tả thuật toán***



***Completeness:*** No

***Optimality:*** No

***Time complexity:*** , tốt hơn DFS

***Space complexity:*** , ít tốn bộ nhớ hơn DFS

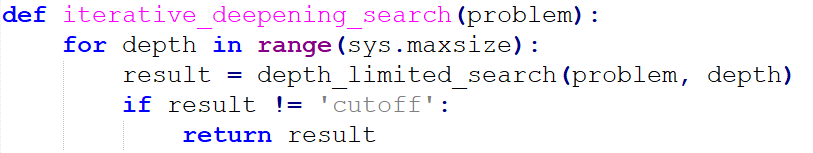
Với l m là depth-limited

## 6.5. Thuật toán Iterative deepening search

***Ý tưởng :*** Thực hiện thuật toán Depth-limited search nhiều lần với giới hạn độ sâu (depth-limited) tăng dần cho đến khi tìm được solution

***Đầu vào :*** problem, đầu ra : solution

***Mô tả thuật toán***



***Completeness:*** Yes

***Optimality:*** Yes

***Time complexity:***

***Space complexity:***

Khi depth-limited tăng dần đến khi bằng hoặc lớn hơn m thì time and space complexity bằng với time and space complexity của DFS.

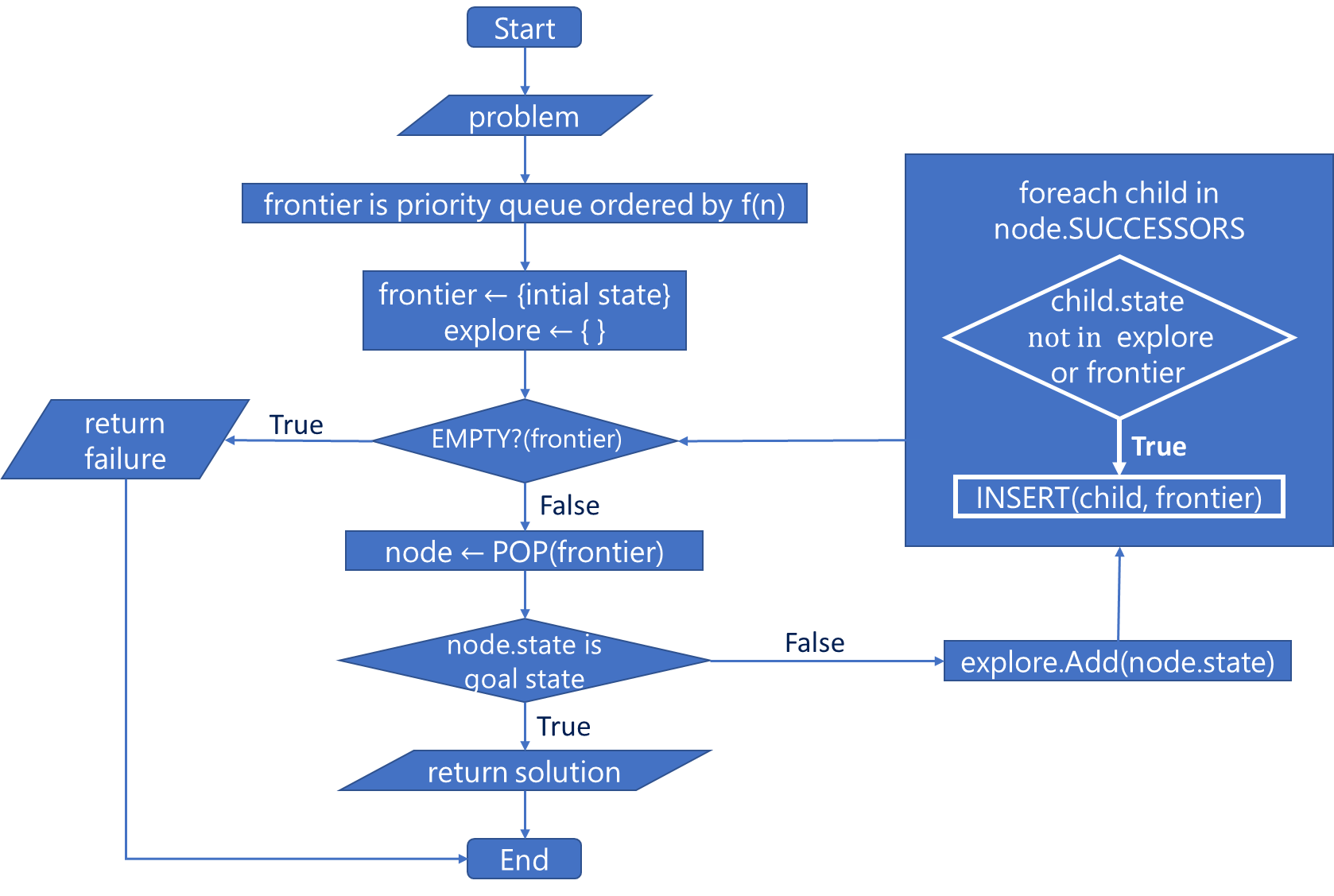
# 7. Các thuật toán Informed search

## 7. 1. Các hàm ước lượng quảng đường

***Evaluation function f(n)*** là hàm sử dụng thông tin có được về môi trường để ước lượng chi phí đường đi từ initial state đến goal state. Giá trị này được dùng để thay đổi chiến lược tìm kiếm hiệu quả hơn.

***Heuristic functions h(n)*** là hàm ước lượng quảng đường đi từ node n đến goal state. Ví dụ : straight line distance heuristic

## 7.2. Ý tưởng chung của Informed search



Hầu hết các thuật toán Informed search đề dựa trên ý tưởng này, chỉ khác nhau ở cách tính f(n)

## 7.3. Thuật toán Best-first search

***Evaluation function:*** chính là heuristic functions f(n) = h(n)

***Đầu vào:*** problem, đầu ra : solution

***Completeness:*** Yes

***Optimality:*** No. Do sử dụng f(n) = h(n) nên Best-first search đã bỏ qua quảng đường từ intial state đến node n (path cost), dẫn đến việc chọn node expand không tối ưu, tìm không được optimal solution

## 7.4. Thuật toán A\* search

***Evaluation function :*** f(n) = h(n) + path cost

***Đầu vào :*** problem, đầu ra : solution

***Completeness:*** Yes

***Optimality:***  Yes

Heuristic functions phải consistent heuristic, tức là h(n) <= cost(n, n’) + h(n’) (bất đẳng thức trong tam giác) với n’ là succesor của node n

Nếu h(n) = 0 -> f(n) = path cost -> A\* = Uniform-cost search (optimal)

***Time complexity:*** Time complexity phụ thuộc vào độ chính xác của heuristic function, nếu giá trị của Heuristic function gần bằng với đường đi thực tế thì tốc độ tìm ra solution càng nhanh. Trong trường hợp xấu nhất, h(n) = 0, time complexity của A\* search bằng với Uniform-cost search là

***Space complexity:***

* Dựa trên Uniform-cost search, mà Uniform-cost search lại dựa trên Breadth-ﬁrst search nên vẫn có thể bị tràn bộ nhớ.
* Để giải quyết vấn đề này, có thể dùng thuật toán Simplified Memory Bounded A\* (SMA). Thuật này loại bỏ những node có f(n) quá lớn

## 7.5. Cách tạo heuristic function

* Cách 1 : Giảm nhẹ vấn đề bằng cách bỏ bớt các hạn chế (restriction) cho mỗi action
* Cách 2 : Dùng cơ sở dữ liệu mẫu (pattern databases)
  + Chọn subproblem (vấn đề con), vẫn giữa nguyên các restriction trên mỗi action
  + Tổ chức pattern databases bằng cách lưu trữ solution costs cho mỗi thể hiện (instance) của subproblem.

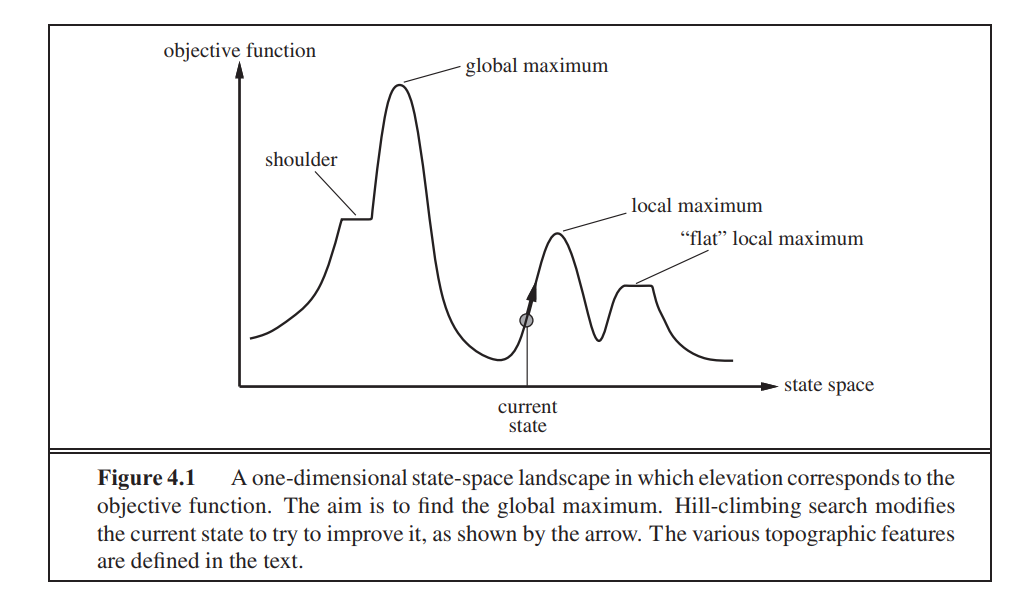
Chọn heuristic function lớn nhất trong các heuristic function vừa tạo ra để có được consistent heuristic

V. LOCAL SEARCH – SEARCHING FOR GOAL STATE

# 1. So sánh với các thuật toán tìm kiếm trước đó:

* Các thuật toán trước đó cần phải biết các thông tin:
  + Obversable: biết rõ được đang ở state nào của môi trường.
  + Discrete: những action ứng với mỗi state của môi trường, là rời rạc
  + Deterministic: mỗi action có 1 hành động duy nhất, nghĩa là mình có thể xác định được khi mình thực hiện action thì mình thu được kết quả gì.
* Các thuật toán phía sau sẽ loại bỏ những ràng buộc trên. Khi đó đòi hỏi phải thay đổi mục tiêu tìm kiếm, cách lưu

# 2. State space landscape:

Local search sẽ tập trung vào 2 thứ: current state và mục đích nó hướng đến (chi phí thấp nhất/phần thưởng nhiều nhất/…). Biểu diễn trực quan dưới dạng:

≫ Objective function = reward function = – cost function

≫ Current state: từ trạng thái hiện tại, ta tìm kiếm các trạng thái kế cận nó.

≫ Khi đạt global optimum, ta nói thuật toán đã optimal.

≫ Optimum = mininum hoặc maximum.

# 3. Local search:

## 3.1. Ý tưởng:

* Cách thức tìm kiếm: tập trung vào các states ở trạng thái hiện tại hoặc gần đó.
* Mục tiêu tìm kiếm: tìm ra goal states, chứ không phải tìm action sequences để đi đến goal states đã xác định. (*Nghĩa là, như trong maze, thì local search sẽ tìm lối thoát của maze ở đâu, chứ không phải tìm đường đi ra khỏi maze*).

## 3.2. Input/Output:

* Input: một problem
* Output: là một goal state hoặc Failure

## 3.3. Completeness:

## 3.4. Optimal:

Không, vì có thể rơi vào local maximum/local minimum

## 3.5. Cost:

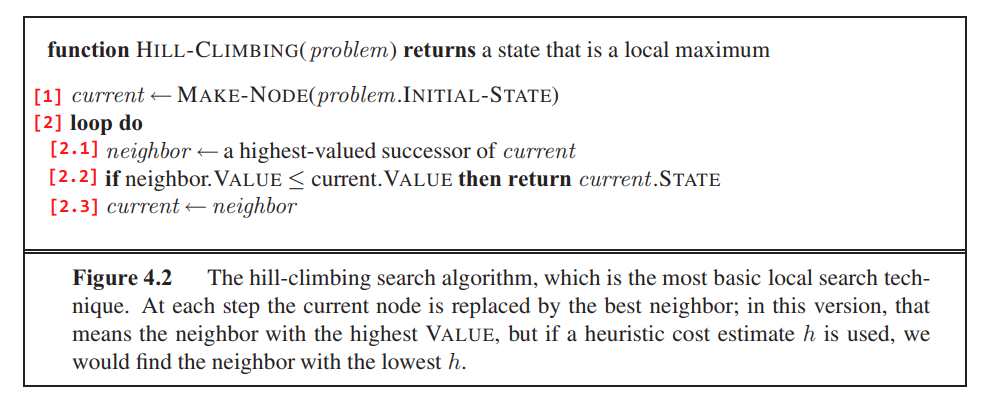
Space: chỉ lưu current node và các neighbors của nó chứ không lưu lại tất cả search tree như những thuật toán trước (*Use only one current node and its neighbors*). Do đó, tiết kiệm không gian bộ nhớ rất nhiều, có thể giải những bài toán continuous, hoặc infinite state space.

# 4. Hill-climbing search:

## 4.1. Ý tưởng:

Chọn best neighbor của current state (để mỗi lần di chuyển, nó luôn có thể “*đi lên*” đồi, nếu không thể đi sang chỗ “*cao hơn*” nữa, thì việc tìm kiếm sẽ dừng lại).

## 4.2. Mô tả thuật toán:

* Input: Một problem với một trạng thái khởi tạo INITIAL-STATE.
* Output: một trạng thái kết quả, là cực đại cục bộ (local maximum).
* ****Pseudocode:
* Giải thích:

[1] Khởi tạo node *current* là node gốc của problem, dựa vào trạng thái khởi tạo ứng với problem đó (*problem*.INITIAL-STATE).

[2] Thực hiện vòng lặp vô tận, cho đến khi tìm ra local maximum thì kết thúc:

[2.1] Chọn neighbor là sucessor có giá trị VALUE lớn nhất trong tất cả các successors của *current* state.

*VALUE là giá trị trả về của hàm objective function (trong state space landscape).*

[2.2] Nếu giá trị VALUE của neighbor không vượt quá current thì trả về trạng thái hiện tại của current (current.STATE).

*Nghĩa là,* từ trạng thái current, không còn cách nào để chọn được một neighbor có giá trị VALUE cao hơn giá *trị* hiện tại – ta không thể “*leo lên*” điểm nào khác “*cao hơn*” nữa (lúc này đã đạt local maximum).

[2.3] *Ngược* lại, xem neighbor là trạng thái mới của current, ta tiếp tục tìm neighbor có giá trị cao nhất ở xung quanh, rồi tiếp tục vòng lặp [2.1].

## 4.3. Completeness:

Incomlete (A hill-climbing algorithm that never makes “downhill” moves toward states with lower value – or higher cost, is gruaranteed to be incomplete, because it can get stuck on a local maximum) [Page 125].

## 4.4. Optimal:

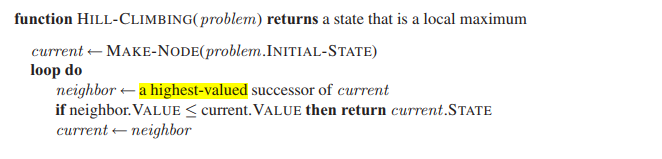
Không, vì không đảm bảo tìm thấy global maximum

## 4.5. Cost:

* Time cost:
* Space cost: chỉ tốn bộ nhớ để lưu trữ giá trị của trạng thái hiện tại, không cần lưu lại tất cả giá trị đã duyệt qua. Do đó tiết kiệm không gian

# 5. Issues of Hill-climbing search:

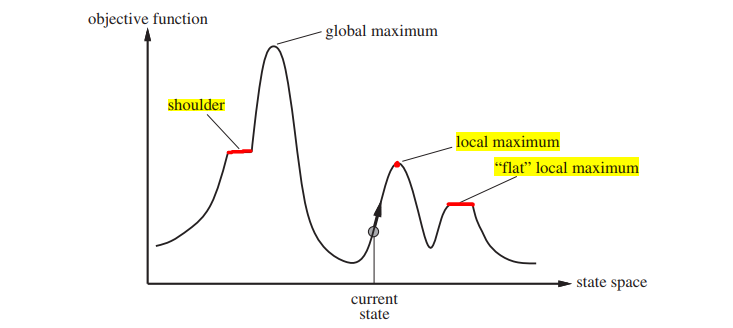
## 5.1. Quá nhiều successors:

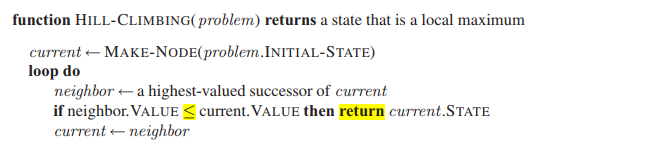
****Sẽ dẫn đến việc tốn nhiều chi phí thời gian (*để tìm max và để tạo các successors*), để xác định neighbor có giá trị VALUE cao nhất.

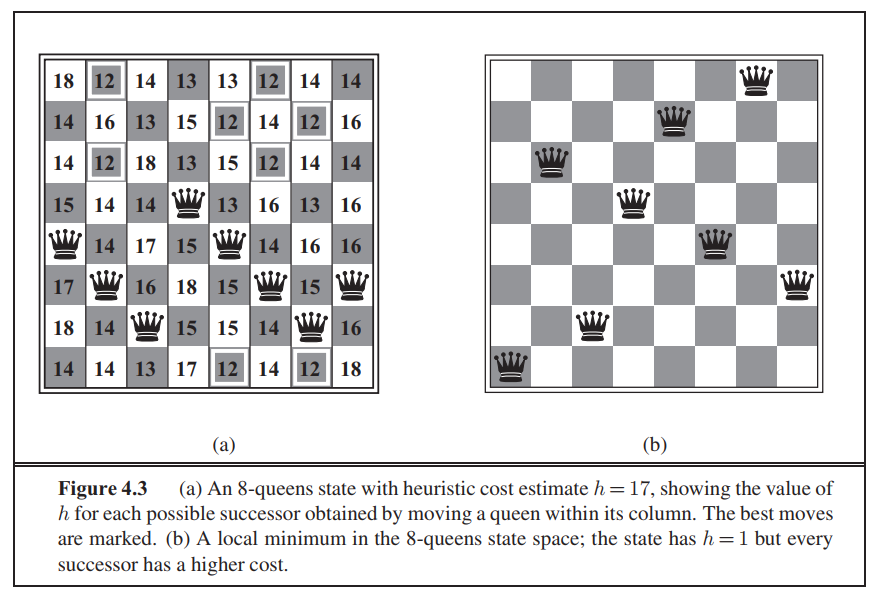
**→** Giải quyết: dùng Stochastic hill-climbing để sinh ngẫu nhiên các succesors cho đến khi nó đạt giá trị VALUE tốt hơn hiện tại. *(không cần phải sinh ra tất cả successors rồi duyệt lần lượt như hill-climbing để tìm successors lớn nhất).*

## 5.2. Local optimum:

Khi current state đạt được local optimum hoặc flateaux (“flat” local optimum hoặc shoulder) (*như hình*). Khi đó, do thuật toán Hill-climbing chỉ lấy điều kiện so sánh *neighbor.VALUE ≤ current.VALUE*, cho nên khi điều kiện trên thỏa mãn thì return và kết thúc quá trình tìm kiếm. Do đó, thuật toán này không optimal.

****

****

Ví dụ: (Hình b) Không thể di chuyển bất kỳ quân hậu nào, để giảm số lượng quân hậu có thể tấn công nhau.

**🡪** Giải quyết:

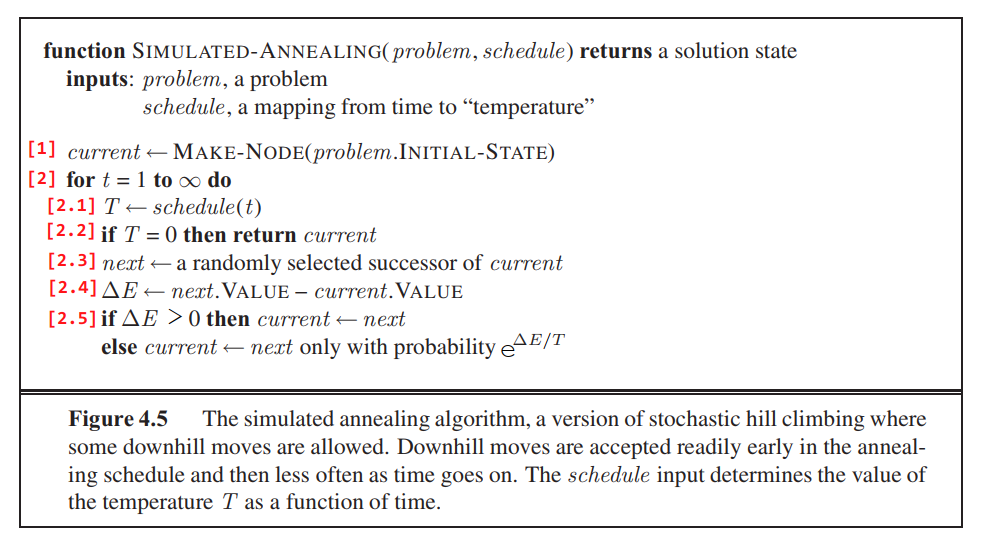
* Để vượt qua các flateaux: cho phép di chuyển ngang **với số bước giới hạn** (*để tránh bị lặp vô tận*), khi gặp neighbor có giá trị VALUE không tăng so với giá trị của current. => Tăng tỉ lệ tìm được lời giải cho bài toán 8-queens từ 14% lên 94%.
* Để giải quyết local optimum:
  + Dùng Random-restart hill-climbing. Nghĩa là trong trường hợp bị rơi vào local optimum, ta sẽ chạy lại thuật toán hill-climbing với giá trị INITIAL\_STATE là một state được chọn random (khác với Initial State ở trường hợp lúc đầu).
  + Dùng Local beam search (*beam: chùm ánh sáng*): (cải tiến của random restart hill-climbing).
  + ***Instead of trying k random searches sequentially, do them*** ***simultaneously.*** Nghĩa là đồng thời thực hiện việc tìm kiếm theo k hướng khác nhau (đa nhiệm), thay vì lượt lượt thử từng cách một, nếu chưa thấy thì restart.
  + ***Local beam search: not just simultaneously, choose best of all searches.*** Không chỉ chọn best neighbor cho mỗi search, mà nó chọn best of all searches (*lấy cái tốt nhất của các best successors thành phần* 🡪 tốn thời gian để tìm max).
  + Stochastic beam search: combines Local beam search and Stochastic hill-climbing. Giải quyết vấn đề này sinh của Local beam search là phải tìm best successor. Khi có quá nhiều successor sẽ dần đến tốn nhiều chi phí cho việc tính Value cho mỗi successor và tìm max.
    - Không tìm max
    - Chọn successor ngẫu nhiên với xác suất dựa theo VALUE

# 6. Simulated Annealing:

## 6.1. Ý tưởng:

## 6.2. Mô tả thuật toán:

* Input: Một problem
* Output: Một state là solution state của bài toán
* Pseudocode:

****

Hình 2 - Russell, S., & Norvig, P. (2016).

* Giải thích:

[1] Khởi tạo node *current* là node gốc của problem, dựa vào trạng thái khởi tạo ứng với problem đó (*problem*.INITIAL-STATE).

[2] Thực hiện vòng lặp vô tận với giá trị khởi tạo của t là 1:

[2.1] Tính giá trị nhiệt độ T = schedule(t)

[2.2] Nếu T=0 thì trả về trạng thái hiện tại của *current*.

[2.3] Ngược lại, chọn giá trị *next* một cách ngẫu nhiên, là một successor của *current.*

[2.4] Tính độ chênh lệch .

[2.5] Kiểm tra xem có tiếp tục vòng lặp đi theo *next* hay không. Chỉ tiếp tục thực hiện nếu (trạng thái next được chọn mang giá trị VALUE tốt hơn trạng thái current), nếu không, chỉ chọn với điều kiện xác suất

## 6.3. Completeness:

## 6.4. Optimal:

## 6.5. Cost:

* Space: chỉ lưu current node và các neighbors của nó chứ không lưu lại tất cả search tree như những thuật toán trước (Use only one current node and its neighbors). Do đó, tiết kiệm không gian bộ nhớ rất nhiều, có thể giải những bài toán continuous, hoặc infinite state space.

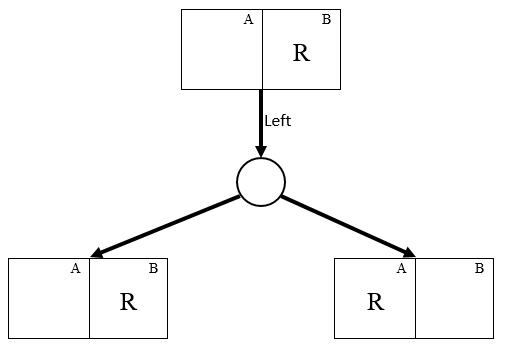
## 6.6. Code, demo:

|  |
| --- |
| def schedule(t, k = 20, lam = 0.005, limit = 1000):  """One possible schedule function for simulated annealing"""  return (k \* np.exp(-lam \* t) if t < limit else 0)  def simulated\_annealing(problem):  # Khởi tạo node hiện tại (current node) của problem  current = Node(problem.initial)  # Vòng lặp 1 đến vô cùng  for t in itertools.count(start = 1):  # Tính giá trị nhiệt độ T  T = schedule(t)  # Nếu nhiệt độ T = 0, quá trình luyện kim (annealing) kết thúc  if T == 0: return current.state  # Tìm các neighbors của current state  successors = current.expand(problem)    # Chọn ngẫu nhiên 1 successor trong tập các successors được expend từ node hiện  # tại (current node)  next = random.choice(successors)  # Tính giá trị deltaE  deltaE = problem.value(next) - problem.value(current)  # Nếu deltaE > 0, nghĩa là nếu chọn node là next, ta có giá trị tốt hơn giá trị  # của current  if deltaE > 0: current = next  # Ngược lại, chọn với điều kiện xác suất e^(deltaE/T)  elif random.uniform(0.0, 1.0) < math.exp(deltaE/T):  current = next |
|  |

# 7. Searching in NONDETERMINISTIC environments

## 7.1. Tìm kiếm trong môi trường không xác định

* Một action có thể trả về nhiều trạng thái
  + Results (State, action) 🡪 States{s1, s2, s3,…}
  + Ví dụ: Robot thực hiện di chuyển sang trái (Vị trí hiện tại là B di chuyển sang A)
    - Results (B, Left) 🡪 States {s1 = B, s2 = A}



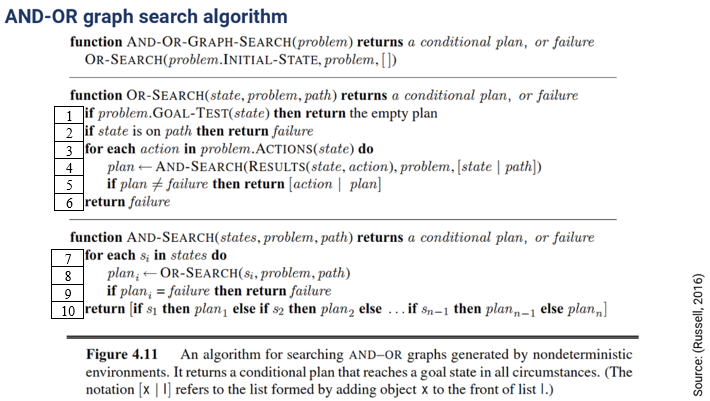
## 7.2. And\_Or Graph Search

***7.2.1. Ý tưởng***

* + Từ một trang thái hiện tại có thể thực hiện các hành động (Or search) 🡪 Chọn một hành động để thực hiện
  + Thực hiện một hành động sẽ cho một hoặc nhiều hơn một trạng thái mới (And search) 🡪 Phải tất cả các trạng thái mới đc tạo ra.
  + Solution là một cây con với tất cả nút lá là trạng thái cần đạt được (ở nút and thì phải bao gồm tất cả các trường hợp)

***7.2.2. Đầu vào, đầu ra***

* + Đầu vào là vấn đề cần giải quyết (problem).
  + Đầu ra là một solution hoặc có thể trả về Failure khi không tìm được giải pháp.

***7.2.3. Thuật toán***

* + Biến
    - state: trạng thái hiện tại (thực hiện or search).
    - problem: vấn đề cần giải quyết.
    - path: mảng chứa các state đã xuất hiện.
    - plan: chứa kết quả trả về (bao gôm action và trạng thái đạt được khi thực hiện action đó).
    - states: chứa nhiều state (thực hiện and search).
  + Hàm xử lý
    - ACTIONS (state) trả về tất cả các hành động khi đang ở state hiện tại.
    - RESULTS (state, action) trả về tất cả state mới khi thực hiện action tại state hiện tại.
  + Chu trình chạy của thuật toán

|  |  |
| --- | --- |
| 1 | Điều kiện thoát đệ quy (thoát nếu là **goal state**) |
| 2 | Điều kiện thoát đệ quy (thoát nếu đã có trong **path**) |
| 3 | Lấy toàn bộ các action có thể thực hiện ở state hiện tại  Thực hiện từng action |
| 4 | Lấy kết quả khi thực hiện action (states 🡨 RESULT (state, action))  Đưa state hiện tại vào path  Gọi **And search** với các states được tạo ra |
| 5  6 | Kiểm tra kết quả trả về của and search  Nếu khác failure thì thêm action vào plan  Nếu trả về failure thì return failure |
| 7 | Lấy ra từng state trong states để thực hiện |
| 8 | Gọi hàm **Or search** với state được lấy ra và lưu kết quả trả về vào **plani** |
| 9 | Kiểm tra **plani** = failure thì return failure (nhánh hiện tại không đi đến kết quả) |
| 10 | Trả về tất cả kết quả của từng state |

***7.2.4. Completeness***

* + Thuật toán completeness

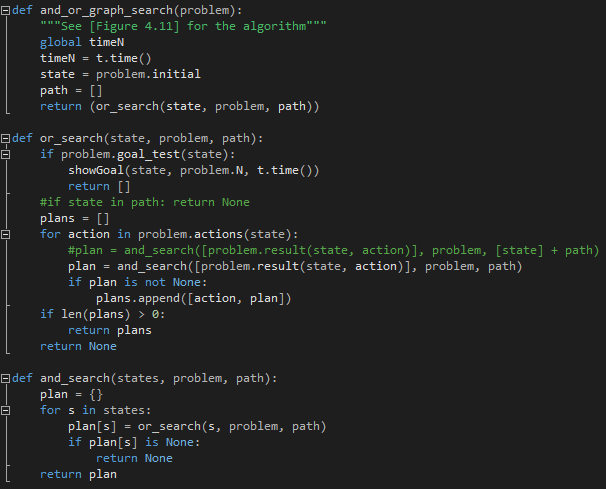
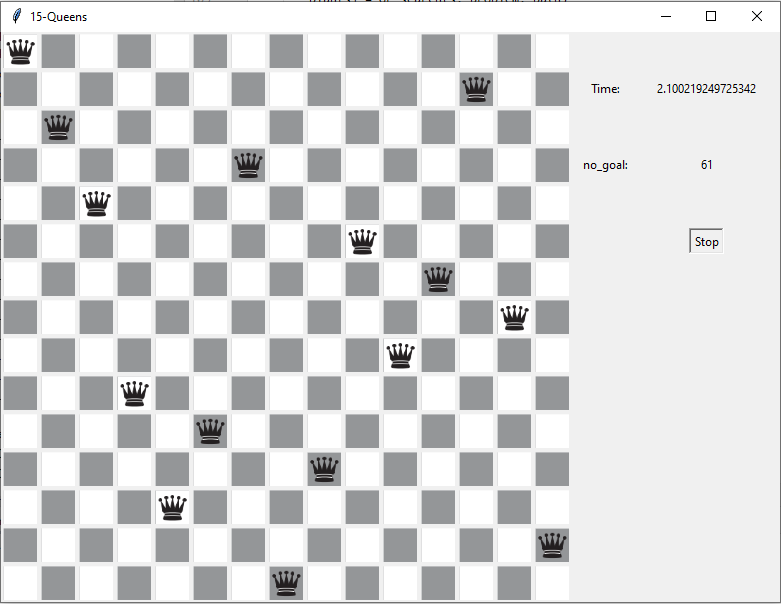
***7.2.5. Optimals***

* + Thuật toán không optimals vì trong thuật toán sử dụng phương pháp tìm kiếm làm DFS (tìm kiếm theo chiêu sâu)

***7.2.6. Cost***

* + Time: tốn ít thơi gian, vị tận dụng được ưu điểm của thuật toán DFS
  + Space: tốn nhiều bộ nhớ, vì sử dụng Graph search phải lưu lại các state đã xuất hiện để kiểm tra.

***7.2.7. Code, Giao diện***

* +  Code
  + Giao Diện

***7.2.8. Ứng dụng thực tế:*** Giải các bài toán thực tế mà không biết rõ tất cả môi trường xung quanh hay không biết kết quả thực sự trả về khi thực hiện hạnh động.

# 8. Searching in Partially Observable Enviroments

* Không biết hoặc quan sát đươc một phần của môi trường
* Ưu điểm:
  + Tiết kiệm được chi phí mua và lắp đặt sensor
  + Tiết kiệm thời gian để sensor lấy thông tin
* Nhược điểm
  + Xử lý không được nhiều bài toán
  + Xử lý vấn đề không hiệu quả
* Belief state: tập hợp các state mà agent có thể đang ở trong thực tế

## 8.1. No observable

***Ý tưởng giải:***

* + Xây dựng belief state
    - initial state: tất cả trạng thái trong thực tế có thể có
    - possible actions: ACTIONS(belief state b) 🡪 b = {s1, s2}

actionsp(s1) = {a, b, c}

actionsp(s2) = {a, c, d}

có thể chọn actions(b) = {a, b, c, d}

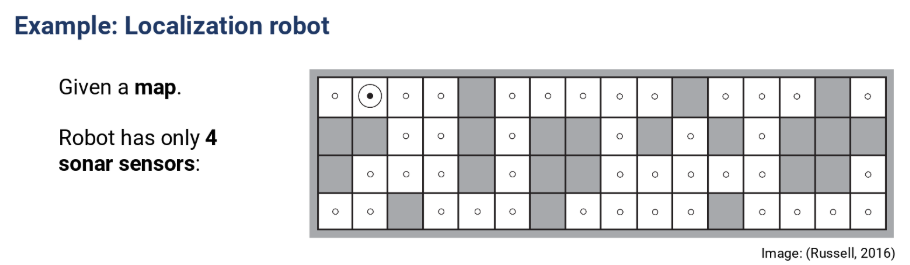
hoặc = {a, b}

* + - Transition model: RESULT (belief state b, action a) = new belief state b’
      * b = {s1, s2}
      * b’ = Resultp (s1, a) ∪ Resultp (s2, a)
    - Goal test: toàn bộ trạng thái trong belief state để là goal state
      * b = {s1, s2}
      * nếu goal-testp(s1) và goal-testp(s2) đều trả về là đúng thì goal-test(b) là đúng
    - cost: step-cost(b) and path-cost(b)
  + Chuyển bài toán thành fully observable
  + Xử dụng các thuật toán BFS, DFS, A\*, AND-OR, … để giải

## 8.2. Partially observable

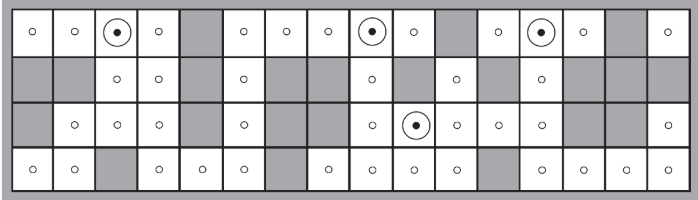
Tương tự như no observable nhưng transition model sẽ được thu hẹp dần nhờ đưa trạng thái belief vào percept kiểm tra nếu thõa điều kiện thì stateb đó được gửi lại trong beilef nếu không thõa thì loại bỏ nó khỏi bilief state.

## 8.3. Demo



Belief state ban đầu là tất cả các vị trí

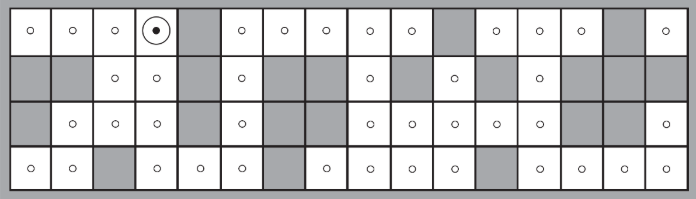
* Step 1: move right



Percept có 1 tường chặn ở trên

🡪Belief state còn lại 4 state

* Step 2: move right



Percept có tường chặn ở trên và bên phải

🡪Belief state còn lại một state

# 9. Online Search

## 9.1. So sánh về offline search và online search :

* Khái niệm offline search và online search
  + Offline search
    - Offline search là các thuật toán chạy trên máy tính trước tiên, để tìm ra solution và đưa cho agent chạy trong thực tế để tìm ra giải pháp. Mỗi action của agent đều có thể biết được kết quả dựa trên transition model.
    - Offline search gồm 5 thành phần : initial state, possible actions, transition model, goal test, cost.
    - Ví dụ : DFS, BFS, A\*,…….
  + Online search
    - Là các thuật toán vừa chạy trong môi trường thực tế vừa học để tìm ra solution, với các bài toán không có transition model. Nếu càng thực hiện thì online search càng thông minh.
    - Online search gồm 4 thành phần : initial state, possible actions, goal test, cost.
    - Ví dụ : Dự báo thời tiết ,…
* Bảng so sánh : (online search sẽ useful trong trường hợp : there is a penalty for, nondeterministic enviroments, no enviroment model)

|  |  |  |
| --- | --- | --- |
|  | Offline search | Online search |
| Thuật toán | Cả offline và online search đều là các thuật toán tìm kiếm | |
| Input | Đưa vào 1 lần ngay khi chương trình bắt đầu | Đưa vào liên tục, sau các hành động thì lại có các input tiếp theo |
| Output | Solution | Plan |
| Thành phần một node | Initial state, possible actions, transition model, goal state, cost | Initial state, possible actions, goal state, cost |
| Cách thức hoạt động | Tìm kiếm solution trên máy tính, đưa ra solution và thực hiện | Đặt agent vào môi trường để agent exploring và tìm solution |
| Hiệu quả | Nhanh hơn, nhưng chỉ giải được các bài toán có đủ 5 thành phần của nó. | Chậm hơn, nhưng có thể giải các bài toán không có transiton model |

## 9.2. Competitive ratio :

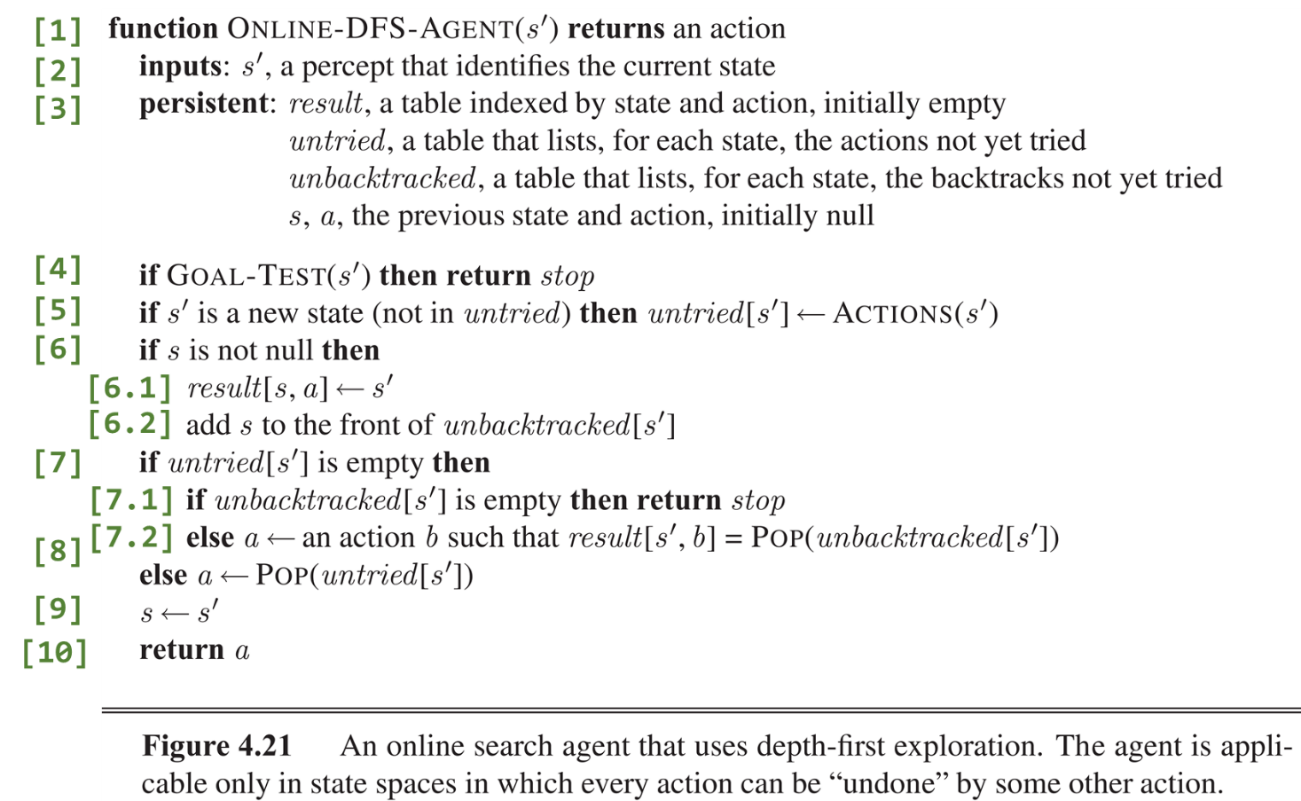
Trong online search có 2 loại của path costs đó là :

* + Actual path cost : cost thực tế mà agent thực hiện, bao gồm các bước explore môi trường và tìm kiếm solution
  + Shortest path cost : cost ngắn nhất để tìm được solution

Competitive ratio = actual path cost / shortest path cost (độ hiệu quả của thuật toán)

## 9.3. Online Depth-first search agent :

* Đầu vào : (s’) percept cho biết cái state hiện tại của enviroment.
* Đầu ra : với mỗi state mà agent nghĩ mình đang ở đó, thì nó nên có action gì, hay nói cách khác là trả về một action đối với 1 state.
* Mô tả thuật toán

**[1]** đầu vào là (s’) state môi trường hiện tại, trả ra với mỗi state thì agent nên có action gì.

**[2]** inputs (s’ : state hiện tại mà agent nghĩ nó đang ở đó)

**[3]** persistent (biến toàn cục)

Result : trả về một bảng chứa (s, a , s’) , có nghĩa là agent đang đứng ở state s và thực hiện action a thì được state s’

Untried : một mảng chứa các state và action mà mình chưa thử qua nó

Unbacktracked : một mảng chứa các state của các trạng thái trước đó mà nó chưa xét tới.

**[4]** Kiểm tra xem (s’) có phải là trạng thái đích chưa, nếu đúng thì dừng lại.

**[5]** Nếu (s’) là một sate mới (không nằm trong untried) thì ta đưa các Action(s’) vào untried[s’].

**[6]** Nếu s không null (nó chỉ null ở lúc bắt đầu) thì ta qua bước 7.

**[6.1]** Ta đưa [s, a] s’ vào bảng result (có nghĩa là ghi được một dòng ở state s thực hiện action a thu được state s’)

**[6.2]** Thêm (s) và unbacktracked[s’]

**[7]** Nếu action của (s’) là rỗng (không có action nào để thực hiện nữa) thì ta qua bước 10.

**[7.1]** Nếu unbacktracked[s’] là rỗng (không có state nào để quay lại thực hiện nữa) thì ta dừng lại.

**[7.2]** Ngược lại, unbacktracked[s’] không là tập rỗng, thì ta thực hiện action b để quay lại trạng thái trước đó để thực hiện (backtrack)

**[8]** Nếu ở [7], action của (s’) không là tập rỗng thì ta lấy ra 1 action từ untried[s’]

**[9]** s’ lúc này sẽ là s.

**[10]** return action a.

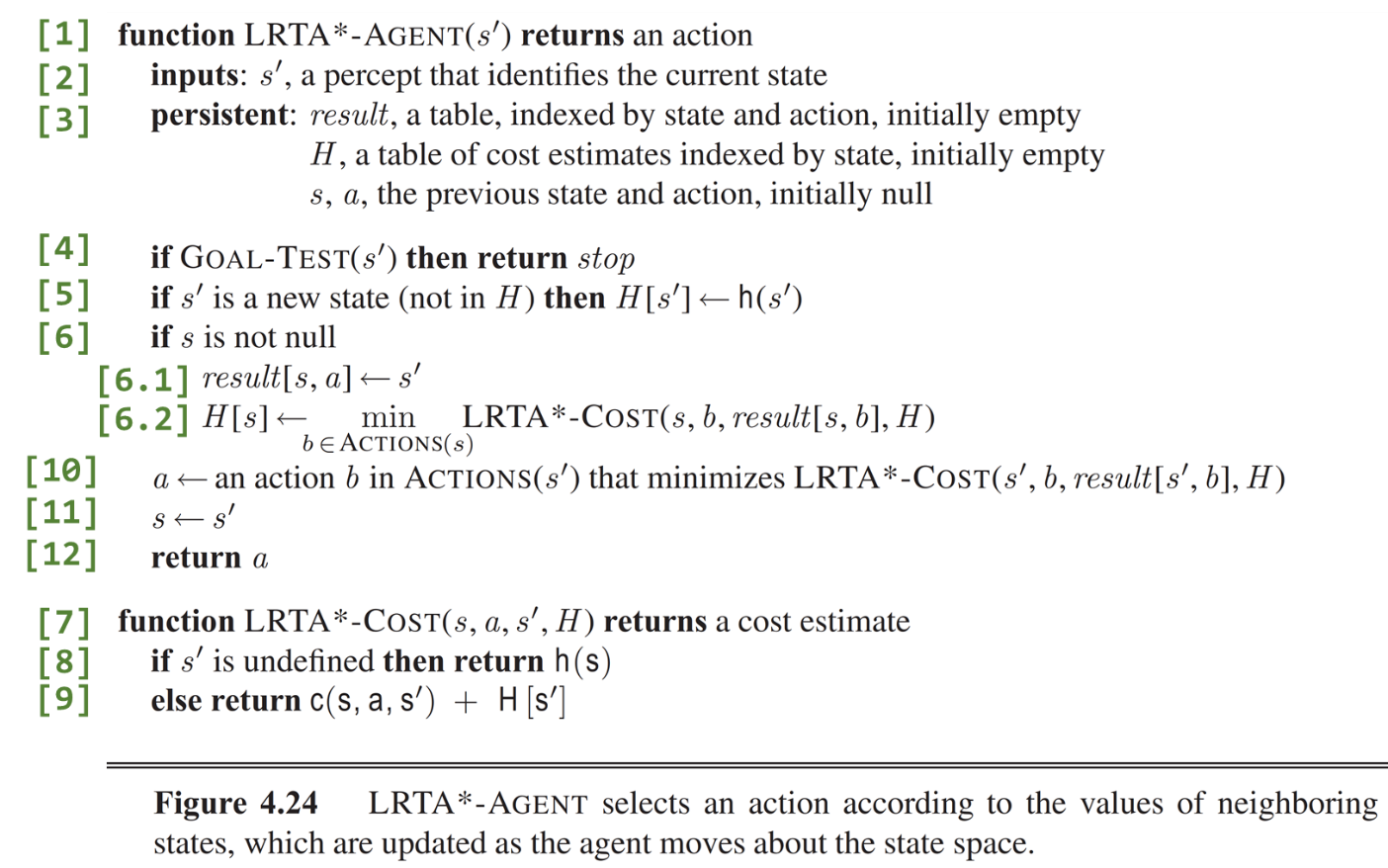
* Completeness : thuật toán sẽ tìm được solution nếu bài toán đó có solution
* Thời gian thực hiện của online search sẽ lâu hơn offline search.

## 9.4. Online A\* search : ( tìm được solution tốt một cách nhanh chóng )

* Ý tưởng : ở từng bước, chúng ta chọn một successor’s theo estimated
* ( ban đầu khỏi tại sẽ là khoảng cách chim hueristic, sau đó agent sẽ vừa đi trong thực tế và vừa cập nhật lại H )

## 9.5. Learning Real – Time A\* agent :

* Đầu vào : (s’) percept cho biết cái state hiện tại của enviroment.
* Đầu ra : với mỗi state mà agent nghĩ mình đang ở đó, thì nó nên có action gì, hay nói cách khác là trả về một action đối với 1 state.
* Mô tả thuật toán



**[1]** Đầu vào là (s’) state hiện tại, kết quả trả về sẽ là một action.

**[2]** inputs ( s’, a)

**[3]** persistent (biến toàn cục)

Result : trả về một bảng chứa (s, a , s’) , có nghĩa là agent đang đứng ở state s và thực hiện action a thì được state s’

H : một bảng ước lượng quảng đường đi từ trạng thái s tới đích, khởi tạo là rỗng

**[4]** Nếu (s’) là trạng thái đích thì chúng ta dừng lại.

**[5]** Nếu (s’) là một state mới, không có trong H, thì ta khởi tạo

**[6]** Nếu s không null (chỉ có trạng thái bắt đầu thì s mới được gán là null)

**[6.1]** Ta đưa [s, a] s’ vào bảng result (có nghĩa là ghi được một dòng ở state s thực hiện action a thu được state s’)

**[6.2]** Cập nhật H[s] bằng giá trị min ở hàm LRTA\*-COST(s, b, result[s, b], H)

( b thuộc Action(s) ), trạng thái s có nhiều action, mỗi action có 1 cost khác nhau, ta chọn ra action có cost thấp nhất.

**[7]** Đầu vào của LRTA\*-COST là, một state s nào đó, thực hiện action a, thu được trạng thái s’ chưa biết.

**[8]** Nếu s’ chưa có trong H thì khởi tạo h(s)

**[9]** Ngược lại nếu nó nằm trong H rồi thì ta cập nhật c(s, a, s’) + H[s’]

**[10]** Chọn action nào đó giúp cho bước đi tiếp theo có cost thấp nhất

**[11]** s’ trở thành s.

**[12]** Trả về action a.

* Completeness : thuật toán sẽ tìm được solution nếu bài toán đó có solution
* Thời gian thực hiện được cải thiện vì nó sử dụng estimated cost

# 10. Constraint Satisfactoin Problems

## 10.1. Constraint satisfaction problems (CSP)

* Atomic state representations : một state tương đương như một thực thể mà mình không thể nhìn thấy và mô tả chi tiết bên trong.
* Factored representation : một state là một tập hợp của các biến
* Constraint satisfaction problems : ( đây là bài toán sử dụng dạng factored representations), tìm ra phép gán cho state để tất cả các giá trị thỏa mãn ràng buộc.
* Trong CSP có 3 components :
  + X ( tập hợp tất cả các biến )
  + D ( tập hợp các giá trị mà các biến được phép nhận )
  + C ( tập hợp các ràng buộc mà các biến phải thỏa mãn )
* Solution của CSP : gán giá trị cho tất cả các biến với giá trị thuộc domains của nó sao cho thỏa mãn tất cả các ràng buộc.

## 10.2. Constraint graph :

* Nodes : là tất cả các biến.
* Links : là 2 biến trong 1 ràng buộc bất kỳ

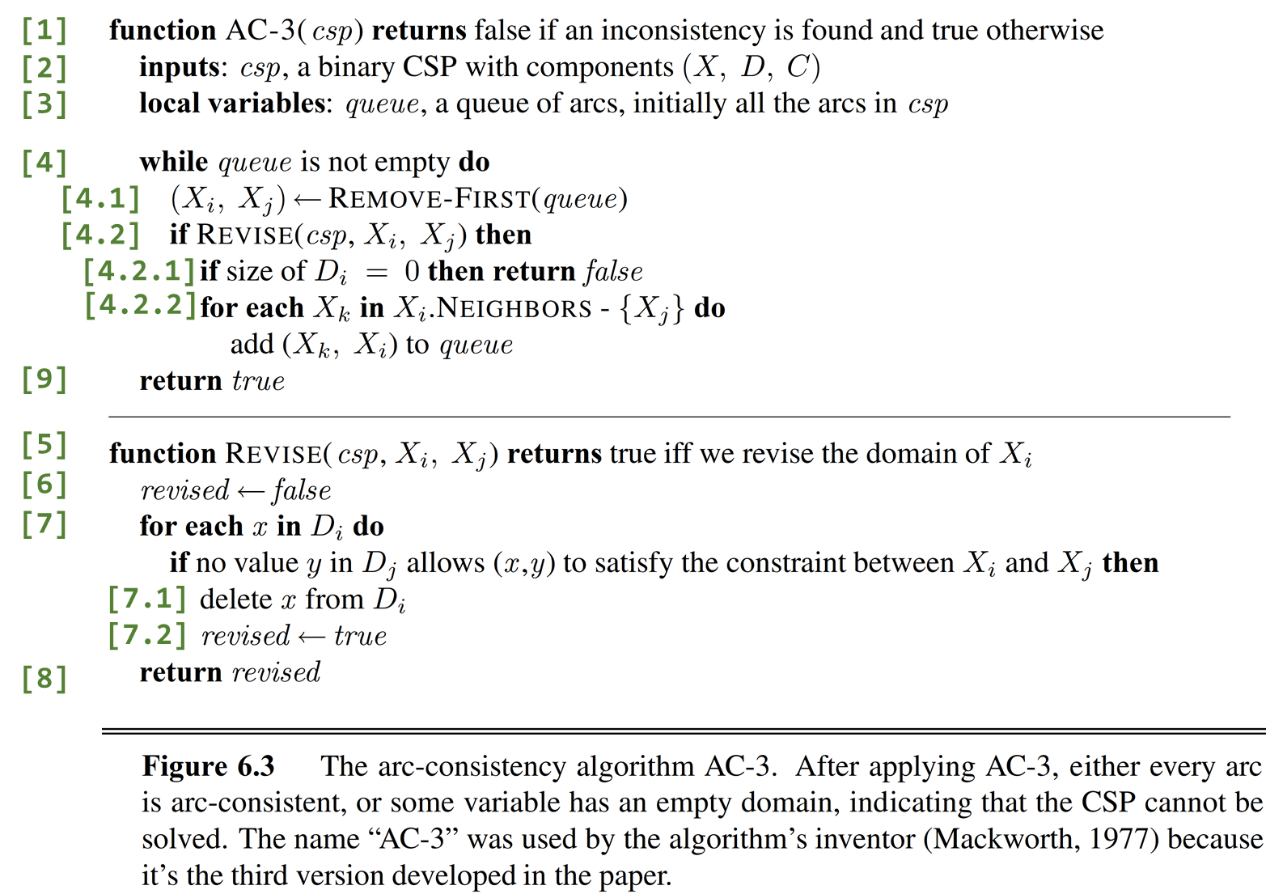
## 10.3. Constraint propagation :

* Ý tưởng : thu hẹp domains của các biến xung quanh nhờ sử dụng các biến có domains là 1 giá trị. (csp giải quyết các bài toán nhanh hơn các thuật toán search thông thường như (BFS, DFS, A\*,…..), trong nhiều trường hợp không gian tìm kiếm quá rộng thì các thuật toán search không tìm ra được nhưng CSP vẫn tìm ra được. )
* Cụ thể : sử dụng những ràng buộc để thu hẹp domain cho các biến, khi một biến nào đó đã được thu hẹp domain thì có thể nó sẽ ảnh hưởng đến domain của biến thứ 2, rồi thứ 3 (sự lan truyền). Đến khi thu hẹp domain cho mỗi biến chỉ còn 1 giá trị thì chúng ta đã tìm được solution.

## 10.4. Local consistency (node , arc , path )

* Consistency : các biến thỏa mãn các ràng buộc thì ta gọi là consisten
* Local consistency : 1 nhóm các biến nằm gần nhau thỏa mãn ràng buộc thì ta gọi là local consistency
* Cách hoạt động của local consistency : mình làm cho các biến nằm gần nhau thỏa mãn hết các ràng buộc, sau đó lan truyền sự thỏa mãn điều kiện đó sang các biến xung quanh, để đảm bảo các biến xung quanh consisten. (loại bỏ những giá trị nằm trong domain làm cho biến đó không consisten )

## 10.5. AC – 3 algorithm (arc consistency)

* Đầu vào : một binary csp của một problem với các components (X, D, C)
* Đầu ra : False nếu không thỏa mãn ràng buộc, True nếu thỏa mãn ràng buộc
* Mô tả thuật toán

**[1]** Đầu vào thuật toán AC-3 của chúng là csp gồm có 3 components (X, D, C)

**[2]** inputs (csp, một binary csp với các components (X, D, C))

**[3]** queue (tập hợp của những constraint có 2 biến)

**[4]** Chạy lần lượt qua tất cả các constraint đó

**[4.1]** Lấy ra 1 constraint 2 biến (Xi, Xj), sau đó đưa (Xi, Xj) vào hàm Revise

**[4.2]** Nếu hàm Revise trả về true ( có thu hẹp domain của Xi) thì ta qua [4.2.1]

**[4.2.1]** Nếu tập domain của Di = 0 (có nghĩa là không có giá trị nào) thì ta return false (bài toán không có solution)

**[4.2.2]** Chọn ra biến Xk  lân cận với biến Xi. Thêm (Xk , Xi) vào queue

**[5]** Hàm Revise nhận đầu vào là (csp, Xi, Xj) và trả về true nếu có thu hẹp domain của Xi, false nếu không thu hẹp được domain của Xi

**[6]** Gán giá trị cho biến revised là false

**[7]** Lấy ra từng giá trị x trong tập domain Di của Xi .Xét nếu không tồn tại giá trị y trong tập domain Dj của Xj , giúp cho (x, y) thỏa mãn ràng buộc thì ta qua [7.1].

**[7.1]** Loại bỏ x khỏi tập domain Di

**[7.2]** Gán cho revised là true

**[8]** Trả về revised

**[9]** Nếu đã thực hiện xét xong các constraint trong queue thì trả về true.

## 10.6. Path consistency :

* Sử dụng cho các constraint 3 biến.
* Chúng ta có một bộ 2 biến { (Xi, Xj) } được xem là path – consistency , khi ta có { Xi = a, Xj = b} sau đó ta tìm được giá trị cho Xm để thoản mãn constraints với {Xi, Xm} và {Xm, Xj}. Còn nếu không tìm được thì tồn tại cặp giá trị (a, b) làm cho nó inconsistent.

## 10.7. K-consistency : (dạng tổng quát của local consistency)

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

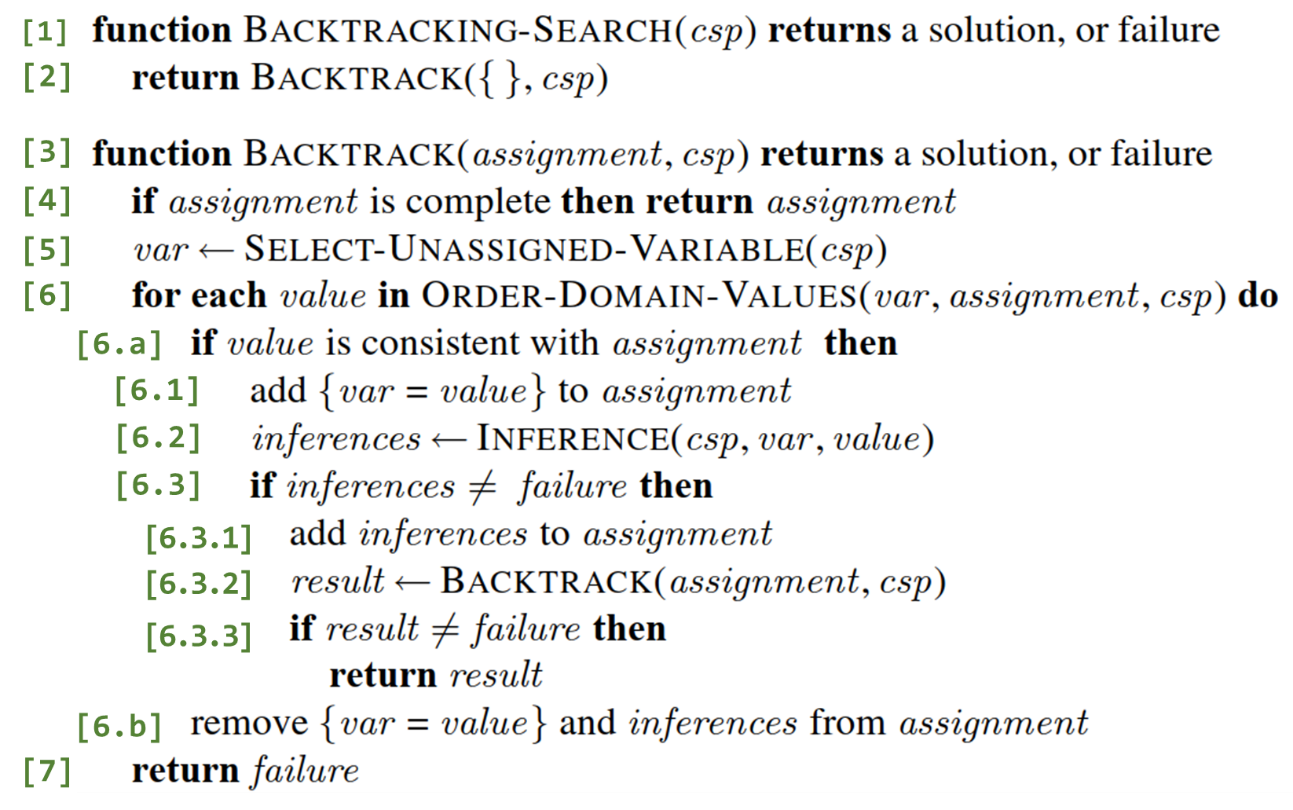
## 10.8. Global constraints :

* Định nghĩa : là một constraint có số lượng biến tùy ý
* Ví dụ : Alldiff constraint, Atmost constraint
  + Alldiff( x1, x2, x3, x4)
  + Với bài toán sodoku 9x9 thì :
  + Alldiff(0,1, 2, 3, 4, 5, 6, 7, 8)

## 10.9. Solving a Sodoku puzzle : (AC – 3, Triplets )

* Triplets :
  + B1 : trong 1 unit, chúng ta tìm ra 3 ô mà domain của 3 ô này là tập con của 1 tập X có 3 giá trị.
  + B2 : loại bỏ 3 số trong tập X ra khỏi domains của các ô còn lại trong unit đó

## 10.10. Backtracking search algorithm

* Ý tưởng : nếu có những biến có domain lớn hơn 1 giá trị, thì ta chọn một giá trị để thử, đưa vào sau đó gán giá trị cho các ô khác, nếu chúng ta gán được cho tất cả các ô thì đã tìm được solution , nhưng nếu chúng ta gán sau 1 số bước thì gặp một ô có domain là rỗng thì không tìm được solution, chứng tỏ giá trị chúng ta chọn ban đầu đã sai, chúng ta loại bỏ giá trị đó khỏi domain của ô đó.
* Các bước chính:
  + B1 : chọn các biến chưa được gán giá trị, thử gán một giá trị cho nó
  + B2 : kiểm tra giá trị đó nếu nó inconsistent với assignment hiện tại thì ta thử một giá trị khác, nếu nó consistent với assignment hiện tại thì ta tiếp tục quay lại B1.
* Bản chất : DFS. Cải tiến hơn DFS , do backtracking sử dụng thuộc tính giao hoán
* Đầu vào : csp
* Đầu ra : solution hoặc failure
* Mô tả thuật toán

**[1]** Đầu vào là một csp gồm có các components (X, D, C) và trả về solution (nếu tất cả các biến đã được gán giá trị) hoặc trả về failure

**[2]** return BackTrack ({}, csp)

**[3]** Hàm BackTrack nhận vào assignment, csp ( đối với bài toán sodoku thì assignment ban đầu là các ô đã có giá trị), (nếu chưa có gì thì đưa tập rỗng vào assignment)

**[4]** Nếu assignment đã hoàn thành, có nghĩa là tất cả các biến đều được gán bằng một giá trị duy nhất, thì assignment đó cũng chính là solution, ta return assignment.

**[5]** Nếu assignment chưa hoàn thành thì ta chọn ra một biến chưa được gán giá trị.

**[6]** Gán cho biến vừa chọn ra một giá trị từ tập domain của nó.

**[6.a]** Nếu giá trị vừa gán cho biến đó consistent với assignment hiện tại thì ta qua [6.1]

**[6.1]** Đưa giá trị đó cho biến và thêm nó vào assignment.

**[6.2]** Kiểm tra xem tập domain của các biến xung quanh có giảm hay không.

**[6.3]** Nếu inferences không gặp biến nào có tập domain là rỗng thì qua [6.3.1]

**[6.3.1]** Thêm inferences vào assignment nếu nó có tập domain chỉ có 1 giá trị

**[6.3.2]** Quay lại [5], lấy biến khác và tiếp tục gán giá trị cho nó.

**[6.3.3]** Nếu result khác failure thì return result

**[6.b]** Nếu ở [6.a] inferences gặp trạng thái failure thì ta remove {var = value} và inferences khỏi assignment và qua [7].

**[7]** return false và qua nhánh khác thực hiện.

## 10.10. Comments on backtracking search algorithm :

* Select next variable and value
  + Chọn biến theo minimum remaining-values heuristic : chọn ra biến có domain nhỏ nhất.
  + Chọn biến theo Degree heuristic : chọn ra biến có số lượng degree lớn nhất ( degree : số lượng constraint mà các biến còn liên quan tới ).
  + Chọn value theo Least-constraning-value heuristic : chọn ra biến để lại cho chúng ta nhiều lựa chọn nhất cho các biến khác.
* Inference
  + Forward checking : kiểm tra domain của các biến lân cận nó
  + Maintaining arc consistency : kiểm tra domain của tất cả các biến
* Backtracking strategies
  + Backtrack theo Backjumping : tạo ta tập conflict-set , chứa các assignments có khả năng xung đột với biến hiện tại mình đang xét.
  + Khi gặp failuer thì ngay lập tức nhảy về tập conflict-set.

TÀI LIỆU THAM KHẢO

* (Ngày 18/12/2017) Lasse Schultebraucks. *A Short History of Artificial Intelligence.* Truy cập ngày 14/6/2020, từ <https://dev.to/lschultebraucks/a-short-history-of-artificial-intelligence-7hm>
* *A Brief History of AI*. Truy cập ngày 14/6/2020, từ <https://aitopics.org/misc/brief-history>
* Wikipedia, *Artificial intelligence*. Truy cập ngày 14/6/2020 từ <https://wikipedia.org/wiki/Artificial_intelligence>
* Wikipedia, *Turing test*. Truy cập ngày 14/6/2020, từ <https://en.wikipedia.org/wiki/Turing_test>
* (Ngày 3/7/2019). *Time complexity of Uniform-cost search*. Truy cập ngày 14/6/2020, từ <https://intellipaat.com/community/3601/time-complexity-of-uniform-cost-search>
* *Time/Space Complexity of Depth First Search*. Truy cập ngày 14/6/2020, từ <https://stackoverflow.com/questions/36479640/time-space-complexity-of-depth-first-search>
* Trần Nhật Quang (2020). *Bài giảng môn Trí tuệ nhân tạo*
* Stuart J. Russell, Peter Norvig (2016). *Artificial intelligence a modern approach.* (Third Edition)