**HCMC UNIVERSITY OF TECHNOLOGY AND EDUCATION**

**FACULTY FOR HIGH QUALITY TRAINING**

****

**Final Project**

**Lecturer : TRẦN NHẬT QUANG**

**Group’s member:**

1. **Nguyễn Đình Thiên Phước 17110062**
2. **Mai Nguyễn Anh Thư 17110082**

**Table of Contents**

[TÌM HIỂU VỀ ĐẠO VĂN 5](#_Toc89334251)

[WEEK 1: INTRODUCTION 5](#_Toc89334252)

[1.1 WHAT IS AI? 5](#_Toc89334253)

[1.2 THE FOUNDATIONS OF ARTIFICIAL INTELLIGENCE 6](#_Toc89334254)

[1.3 THE HISTORY OF ARTIFICIAL INTELLIGENCE 6](#_Toc89334255)

[1.4 RISKS AND BENEFITS OF AI 9](#_Toc89334256)

[WEEK 2: INTRODUCTION TO PYTHON PROGRAMMING 11](#_Toc89334257)

[1. INSTALL 11](#_Toc89334258)

[2. PYTHON BASICS 11](#_Toc89334259)

[2.1 Creating Variables 11](#_Toc89334260)

[2.2 Python Conditions and If statements 11](#_Toc89334261)

[2.3 Python Loops 12](#_Toc89334262)

[2.4 Python Functions 12](#_Toc89334263)

[WEEK 4 : ETHICS AND LAWS FOR AI 13](#_Toc89334264)

[WEEK 5 BASICS CONCEPTS 14](#_Toc89334265)

[1. AGENTS 14](#_Toc89334266)

[2. THE NATURE OF ENVIRONMENTS 15](#_Toc89334267)

[2.1. Fully observable environments vs Partially Observable 15](#_Toc89334268)

[2.2. Single-agent vs Multi-agent 15](#_Toc89334269)

[2.3. Deterministic vs Stochastic 15](#_Toc89334270)

[2.4. Competitive vs Collaborative 15](#_Toc89334271)

[2.5. Dynamic vs Static 15](#_Toc89334272)

[2.6. Discrete vs Continuous 16](#_Toc89334273)

[3. THE STRUCTURE OF AGENTS 16](#_Toc89334274)

[3.1 Agent programs 16](#_Toc89334275)

[3.2 Simple reflex agents 17](#_Toc89334276)

[3.3 Model-based reflex agents 18](#_Toc89334277)

[3.4 Goal-based agents 18](#_Toc89334278)

[3.5. Utility-based agents 19](#_Toc89334279)

[3.6 Learning Agents 20](#_Toc89334280)

[4. PROBLEM-SOLVING AGENTS 21](#_Toc89334281)

[WEEK 6: SEARCHING FOR SOLUTIONS 23](#_Toc89334282)

[1. SEARCHING FOR SOLUTIONS 23](#_Toc89334283)

[2. TREE SEARCH AND GRAPH SEARCH 23](#_Toc89334284)

[2.1 Tree search 23](#_Toc89334285)

[2.2 Graph Search 24](#_Toc89334286)

[3. Uninformed Search Algorithms: 24](#_Toc89334287)

[3.1 Breadth First Search: 25](#_Toc89334288)

[3.2 Uniform-cost search algorithm 26](#_Toc89334289)

[3.3 Depth First Search: 27](#_Toc89334290)

[WEEK 7: INFORMED SEARCH ALGORITHMS 28](#_Toc89334291)

[1. GREEDY BEST-FIRST SEARCH 28](#_Toc89334292)

[2. A\* SEARCH ALGORITHM 29](#_Toc89334293)

[3. HEURISTIC FUNCTIONS 30](#_Toc89334294)

[3.1 The effect of heuristic accuracy on performance 30](#_Toc89334295)

[3.2. Generating admissible heuristics from relaxed problems 31](#_Toc89334296)

[3.3. Generating admissible heuristics from subproblems: Pattern databases 31](#_Toc89334297)

[WEEK 8 : SEARCHING IN MORE COMPLEX ENVIRONMENTS 32](#_Toc89334298)

[1. LOCAL SEARCH 32](#_Toc89334299)

[2. LOCAL SEARCH OTHER 34](#_Toc89334300)

[2.1 Local beam search 34](#_Toc89334301)

[2.2 Simulated annealing 35](#_Toc89334302)

[WEEK 9: SEARCHING IN NONDETERMINISTIC ENVIRONMENTS 36](#_Toc89334303)

[1 AND–OR SEARCH TREES 36](#_Toc89334304)

[2. SEARCHING IN PARTIALLY OBSERVABLE ENVIROMENTS 37](#_Toc89334305)

[2.1 Solving partially observable problems 38](#_Toc89334306)

[2.2 An agent for partially observable environments 39](#_Toc89334307)

[WEEK 10: SEARCHING WITH NO TRANSITION MODEL – ONLINE SEARCH 41](#_Toc89334308)

[1. ONLINE SEARCH 41](#_Toc89334309)

[1.1 Online search problems 41](#_Toc89334310)

[2 Online search agents 41](#_Toc89334311)

[3 Online local search 42](#_Toc89334312)

[2. CONSTRAINT SATISFACTOIN PROBLEMS 44](#_Toc89334313)

[2.1 Definition 44](#_Toc89334314)

[2.2 Constraint propagation: inference in CSPS 44](#_Toc89334315)

[WEEK 11: BACKTRACKING SEARCH FOR CSPS 47](#_Toc89334316)

[1 Variable and value ordering 47](#_Toc89334317)

[2 Interleaving search and inference 48](#_Toc89334318)

[3 Intelligent backtracking: Looking backward 48](#_Toc89334319)

[4 Constraint learning 48](#_Toc89334320)

[WEEK 12 SOLVE CSPS 49](#_Toc89334321)

[1. SOLVE CSPS USING LOCAL SEARCH 49](#_Toc89334322)

[2. Solve CSPs using Constraint Graphs 49](#_Toc89334323)

[2.1 Tree-structured CSP 49](#_Toc89334324)

[WEEK 13 INTRODUCE REINFORCEMENT LEARNING 51](#_Toc89334325)

[1. INTRODUCTION 51](#_Toc89334326)

[1.1 Supervised learning 51](#_Toc89334327)

[1.2 Unsupervised learning 51](#_Toc89334328)

[1.3 Reinforcement learning 51](#_Toc89334329)

[2 CROSS-ENTROPY METHOD 52](#_Toc89334330)

# TÌM HIỂU VỀ ĐẠO VĂN

• Ăn cắp và chuyển giao (ý tưởng hoặc lời nói của người khác) như của riêng mình

• Sử dụng (sản xuất của người khác) mà không ghi nguồn

• Trình bày dưới dạng mới và nguyên bản một ý tưởng hoặc sản phẩm có nguồn gốc từ một nguồn hiện có

Những điều không nên làm

• Tạo trích dẫn sai cho các ý tưởng 'ghi có' không phải của riêng bạn

• Trích dẫn lời của ai đó mà không thừa nhận họ

• Sao chép hoặc mua một bài nghiên cứu / thuật ngữ và biến nó thành của riêng bạn

• Sử dụng các từ chính xác của người khác trong tác phẩm của bạn mà không trích dẫn nguồn hoặc ghi có tác giả

• Diễn giải hoặc tái cấu trúc ý tưởng trong khi phụ thuộc quá nhiều vào tác phẩm gốc của tác giả

Những điều nên làm:

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

• Ghi nguồn tham khảo theo đúng quy định trích dẫn.

• Nắm được những gì không cần trích dẫn

• Hãy tìm hiểu kỹ về vấn đề mà bạn đang muốn nói tới.

* ***Chúng em xin cam đoan đồ án này do các thành viên trong nhóm thực hiện. Chúng em không sao chép, sử dụng bất kỳ tài liệu, mã nguồn… của người khác mà không ghi nguồn. Chúng em xin chịu hoàn toàn trách nhiệm nếu vi phạm đạo văn.***
* Nếu vi phạm đạo văn, đạo code sẽ bị phạt tùy theo mức độ vi phạm, ví dụ trừ điểm nhóm, rớt môn…

# WEEK 1: INTRODUCTION

## **1.1 WHAT IS AI?**

Artificial intelligence (AI) is intelligence demonstrated by machines, as opposed to natural intelligence displayed by animals including humans. Leading AI textbooks define the field as the study of "intelligent agents": any system that perceives its environment and takes actions that maximize its chance of achieving its goals. Some popular accounts use the term "artificial intelligence" to describe machines that mimic "cognitive" functions that humans associate with the human mind, such as "learning" and "problem solving", however, this definition is rejected by major AI researchers

## **1.2 THE FOUNDATIONS OF ARTIFICIAL INTELLIGENCE**

**- Philosophy**

+ Can formal rules be used to draw valid conclusions?

+ How does the mind arise from a physical brain?

+ Where does knowledge come from?

+ How does knowledge lead to action?

**- Mathematics**

+ What are the formal rules to draw valid conclusions?

+ What can be computed?

+ How do we reason with uncertain information?

**- Economics**

+ How should we make decisions in accordance with our preferences?

+ How should we do this when others may not go along?

+ How should we do this when the payoff may be far in the future?

- **Neuroscience**

+ How do brains process information?

- **Psychology**

**+** How do humans and animals think and act?

+ Computer engineering

+ How can we build an efficient computer?

+ Control theory and cybernetics

+ How can artifacts operate under their own control?

- **Linguistics**

+ How does language relate to thought?

## **1.3 THE HISTORY OF ARTIFICIAL INTELLIGENCE**

**1.3.1 The gestation of artificial intelligence (1943–1955)**

- Warren McCulloch and Walter Pitts (1943) Inspired by the mathematical modeling work of Pitts' mentor Nicolas(1936, 1938), they drew upon three sources: knowledge of the basic physiology and function of nerve cells in the brain; a formal analysis of Russell and Whitehead's propositional logic

- Two undergraduate students at Harvard, Marvin Minsky (1927–2016) and Dean Edmonds, built the first neural network computer in 1950

- In 1955, John McCarthy of Dartmouth College convinced Minsky, Claude Shannon, and Nathaniel Rochester to help him bring together U.S. researchers interested in automata theory, neural nets, and the study of intelligence

**1.3.2 Early enthusiasm, great expectations (1952–1969)**

**-** The success of GPS and subsequent programs as models of cognition led Newell and Simon (1976) to formulate the famous physical symbol system hypothesis, which states that “a physical symbol system has the necessary and sufficient means for general intelligent action

- At IBM, Nathaniel Rochester and his colleagues produced some of the first AI programs. Herbert Gelernter (1959) constructed the Geometry Theorem Prover, which was able to prove theorems that many students of mathematics would find quite tricky. This work was a precursor of modern mathematical theorem provers.

- In 1958, John McCarthy made two important contributions to AI. In MIT AI Lab Memo No.1, he defined the high-level language Lisp, which was to become the dominant AI programming language for the next 30 years

**1.3.3 A dose of reality (1966–1973)**

- The failure of Herbert Simon: that within 10 years a computer would be chess champion and a significant mathematical theorem would be proved by machine. These predictions came true (or approximately true) within 40 years rather than 10. Simon’s overconfidence was due to the promising performance of early AI systems on simple examples. In almost all cases, however, these early systems failed on more difficult problems.

- The new back-propagation learning algorithms that were to cause an enormous

resurgence in neural-net research in the late 1980s and again in the 2010s had already been developed in other contexts in the early 1960s (Kelley, 1960; Bryson, 1962)

**1.3.4 Expert systems (1969–1986)**

**-**  The first successful commercial expert system, R1, began operation at the Digital Equipment Corporation (McDermott, 1982). The program helped configure orders for new computer systems

- In 1981, the Japanese government announced the “Fifth Generation” project, a 10-year plan to build massively parallel, intelligent computers running Prolog.

**-** Overall, the AI industry boomed from a few million dollars in 1980 to billions of dollars in 1988, including hundreds of companies building expert systems, vision systems, robots, and software and hardware specialized for these purposes.

**1.3.5 The return of neural networks (1986–present)**

**-** In the mid-1980s at least four different groups reinvented the back-propagation learning algorithm first developed in the early 1960s. The algorithm was applied to many learning problems in computer science and psychology, and the widespread dissemination of the results in the collection Parallel Distributed Processing (Rumelhart and McClelland, 1986)caused great excitement.

- Geoff Hinton, a leading figure in the resurgence of neural networks in the 1980s and 2010s, has described symbols as the “luminiferous aether of AI”—a reference to the non-existent medium through which many 19th-century physicists believed that electromagnetic waves propagated

**1.3.6 Probabilistic reasoning and machine learning (1987–present)**

- 1988 the connection between AI and other fields, including statistics, operations research, decision theory, and control theory. Judea Pearl’s (1988) Probabilistic Reasoning in Intelligent Systems led to a new acceptance of probability and decision theory in AI. Pearl’s development of Bayesian networks yielded a rigorous and efficient formalism for representing uncertain knowledge as well as practical algorithms for probabilistic reasoning

**1.3.7 Big data (2001–present)**

- Banko and Brill (2001) argued that the improvement in performance obtained from increasing the size of the data set by two or three orders of magnitude outweighs any improvement that can be obtained from tweaking the algorithm

- Hays and Efros (2007) developed a clever method for doing this by blending in pixels from similar images; they found that the technique worked poorly with a database of only thousands of images but crossed a threshold of quality with millions of images

**1.3.8 Deep learning (2011–present)**

- In the 2012 ImageNet competition, which required classifying images into one of a thousand categories (armadillo, bookshelf, corkscrew, etc.), a deep learning system created in Geoffrey Hinton’s group at the University of Toronto (Krizhevsky et al., 2013) demonstrated dramatic improvement over previous systems, which were based largely on handcrafted features. Since then, deep learning systems have exceeded human performance on some vision tasks (and lag behind in some other tasks). Similar gains have been reported in speech recognition, machine translation, medical diagnosis, and game playing. The use of a deep network to represent the evaluation function contributed to ALPHAGO’s victories over the leading human Go players (Silver et al., 2016, 2017, 2018).

## **1.4 RISKS AND BENEFITS OF AI**

Some of these are already apparent, while others seem likely based on current trends:

+ LETHAL AUTONOMOUS WEAPONS: These are defined by the United Nations as weapons that can locate, select, and eliminate human targets without human intervention.

+ SURVEILLANCE AND PERSUASION: While it is expensive, tedious, and sometimes legally questionable for security personnel to monitor phone lines, video camera feeds, emails, and other messaging channels, AI (speech recognition, computer vision, and natural language understanding) can be used in a scalable fashion to perform mass surveillance of individuals and detect activities of interest

**+** BIASED DECISION MAKING: Careless or deliberate misuse of machine learning algorithms for tasks such as evaluating parole and loan applications can result in decisions that are biased by race, gender, or other protected categories.

+ IMPACT ON EMPLOYMENT: Concerns about machines eliminating jobs are centuries old.

+ SAFETY-CRITICAL APPLICATIONS: As AI techniques advance, they are increasingly used in high-stakes, safety-critical applications such as driving cars and managing the water supplies of cities.

+ CYBERSECURITY: AI techniques are useful in defending against cyberattack, for example by detecting unusual patterns of behavior, but they will also contribute to the potency, survivability, and proliferation capability of malware

# WEEK 2: INTRODUCTION TO PYTHON PROGRAMMING

## **1. INSTALL**

* Install python

Link download : <https://www.python.org/downloads/>

* Install an IDE or editor (recommend: Visual Studio Code or Visual Studio Community)

Link download : <https://code.visualstudio.com/download>

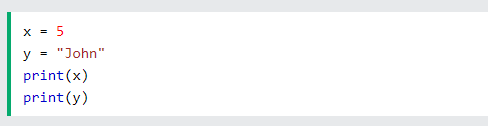
## **2. PYTHON BASICS**

### **2.1 Creating Variables**

Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

Example



### **2.2 Python Conditions and If statements**

Python supports the usual logical conditions from mathematics:

* Equals: a == b
* Not Equals: a != b
* Less than: a < b
* Less than or equal to: a <= b
* Greater than: a > b
* Greater than or equal to: a >= b

These conditions can be used in several ways, most commonly in "if statements" and loops. An "if statement" is written by using the if keyword.

* **Elif**

The elif keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

* **Else**

The else keyword catches anything which isn't caught by the preceding conditions.

* 1. **Python Loops**

Python has two primitive loop commands:

- While loops: we can execute a set of statements as long as a condition is true.

* With the **break** statement we can stop the loop even if the while condition is true
* With the continue statement we can stop the current iteration, and continue with the next
* With the **else** statement we can run a block of code once when the condition no longer is true

- For loops: we can execute a set of statements, once for each item in a list, tuple, set etc.

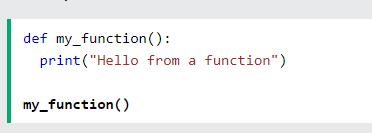
* With the **break** statement we can stop the loop before it has looped through all the items
* With the continue statement we can stop the current iteration of the loop

### **2.4 Python Functions**

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

In Python a function is defined using the **def** keyword

To call a function, use the function name followed by parenthesis



# WEEK 4 : ETHICS AND LAWS FOR AI

**1. AI for ethics and law**

This theme is about applying AI techniques to ethical or legal problems faced by humans. It addresses the development of AI-support for human ethical or legal decision making, investigation and compliance.

* Symbolic approaches include knowledge-based systems, argumentation models and multi-agent systems.
* Data-driven approaches apply machine learning to such tasks as text analytics, predicting outcomes of legal cases, predictive policing, discovering fraud or identifying citizens who are in need of support.
* Combinations of symbolic and data-driven approaches are studied, for instance, for explaining the results of ‘black-box’ machine-learning applications. In the law this is particularly important, since receiving an explanation of a legal decision is one of the fundamental rights of citizens, embodied in many legal procedures and regulatory frameworks.

**2. Responsible autonomous systems**

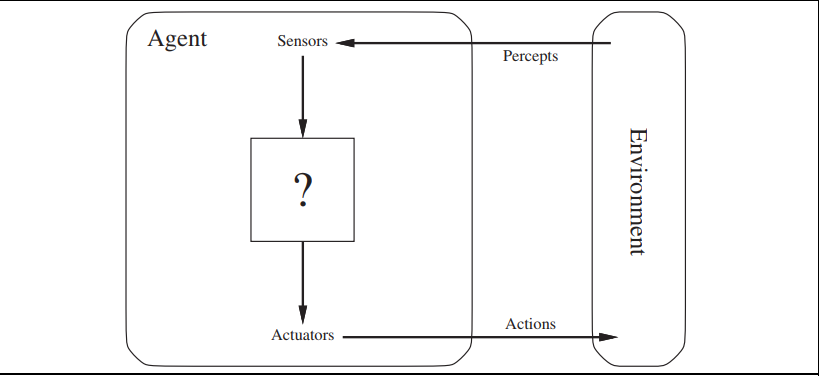
This theme is also about applying AI techniques to ethical or legal problems, but this time for making automated autonomous systems behave in ethically and legally responsible ways. Increasingly, computer systems with some degree of autonomy are being employed in practice. Such artificially intelligent software can do things that, when done by humans, are regulated by law.

For example, self-driving cars have to obey the traffic laws, online information systems have to comply with data protection law, care robots can damage property or the health of the persons they care for, and autonomous weapons have to comply with the laws of war. This raises the problem of how autonomous systems can be designed in such a way that their behavior complies with ethical principles or the law.

# WEEK 5 BASICS CONCEPTS

## **1. AGENTS**

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators. This simple idea is illustrated in Figure 5.1. A human agent has eyes, ears, and other organs for sensors and hands, legs, vocal tract, and so on for actuators. A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators. A software agent receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.



## **2. THE NATURE OF ENVIRONMENTS**

Types of environments

### **2.1. Fully observable environments vs Partially Observable**

* When an agent sensor is capable to sense or access the complete state of an agent at each point in time, it is said to be a fully observable environment else it is partially observable.
* Maintaining a fully observable environment is easy as there is no need to keep track of the history of the surrounding.
* An environment is called unobservable when the agent has no sensors in all environments.

### **2.2. Single-agent vs Multi-agent**

* An environment consisting of only one agent is said to be a single-agent environment.
* A person left alone in a maze is an example of the single-agent system.
* An environment involving more than one agent is a multi-agent environment.

### **2.3. Deterministic vs Stochastic**

* When a uniqueness in the agent’s current state completely determines the next state of the agent, the environment is said to be deterministic.
* The stochastic environment is random in nature which is not unique and cannot be completely determined by the agent.

### **2.4. Competitive vs Collaborative**

* An agent is said to be in a competitive environment when it competes against another agent to optimize the output.
* The game of chess is competitive as the agents compete with each other to win the game which is the output.
* An agent is said to be in a collaborative environment when multiple agents cooperate to produce the desired output.

### **2.5. Dynamic vs Static**

* An environment that keeps constantly changing itself when the agent is up with some action is said to be dynamic.
* A roller coaster ride is dynamic as it is set in motion and the environment keeps changing every instant.
* An idle environment with no change in its state is called a static environment.
* An empty house is static as there’s no change in the surroundings when an agent enters.

### **2.6. Discrete vs Continuous**

* If an environment consists of a finite number of actions that can be deliberated in the environment to obtain the output, it is said to be a discrete environment.
* The game of chess is discrete as it has only a finite number of moves. The number of moves might vary with every game, but still, it’s finite.
* The environment in which the actions performed cannot be numbered ie. is not discrete, is said to be continuous.
* Self-driving cars are an example of continuous environments as their actions are driving, parking, etc. which cannot be numbered.

## **3. THE STRUCTURE OF AGENTS**

The job of AI is to design an agent program that implements the agent function the mapping from percepts to actions. We assume this program will run on some sort of computing device with physical sensors and actuators we call this the architecture:

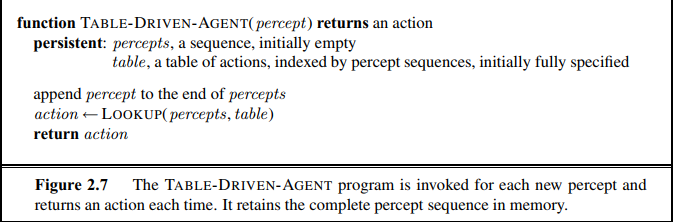
**agent = architecture + program**

### **3.1 Agent programs**

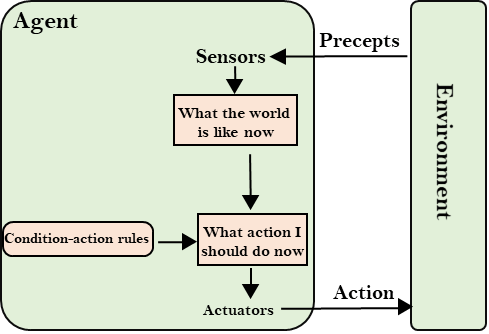
The agent programs that we design in this book all have the same skeleton: they take the current percept as input from the sensors and return an action to the actuators.4 Notice the difference between the agent program, which takes the current percept as input, and the agent function, which takes the entire percept history

We describe the agent programs in the simple pseudocode language that is defined in

Appendix B



* 1. **Simple reflex agents**
* The Simple reflex agents are the simplest agents. These agents take decisions on the basis of the current percepts and ignore the rest of the percept history.
* These agents only succeed in the fully observable environment.
* The Simple reflex agent does not consider any part of percepts history during their decision and action process.
* The Simple reflex agent works on Condition-action rule, which means it maps the current state to action. Such as a Room Cleaner agent, it works only if there is dirt in the room.
* Problems for the simple reflex agent design approach:
* They have very limited intelligence
* They do not have knowledge of non-perceptual parts of the current state
* Mostly too big to generate and to store.
* Not adaptive to changes in the environment.



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

- The Model-based agent can work in a partially observable environment, and track the situation.

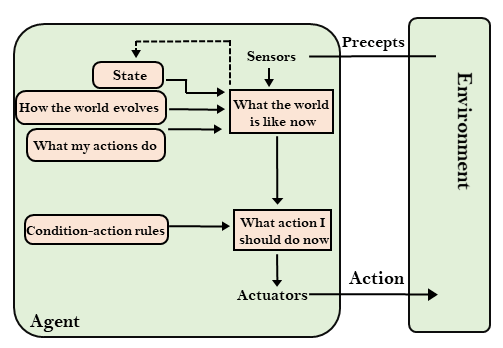
- A model-based agent has two important factors:

* Model: It is knowledge about "how things happen in the world," so it is called a Model-based agent.
* Internal State: It is a representation of the current state based on percept history.

- These agents have the model, "which is knowledge of the world" and based on the model they perform actions.

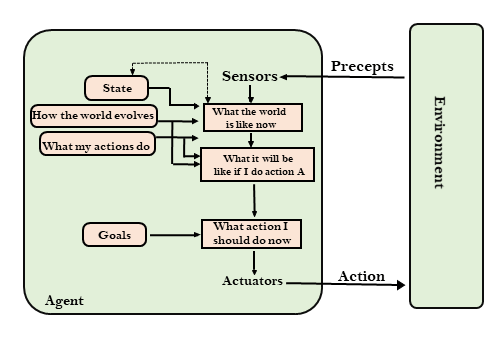
- Updating the agent state requires information about:

* How the world evolves
* How the agent's action affects the world.



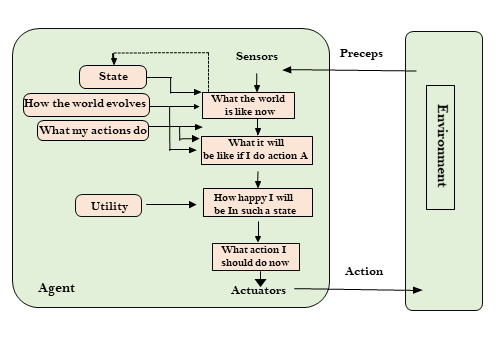
### **3.4 Goal-based agents**

* The knowledge of the current state environment is not always sufficient to decide for an agent to what to do.
* The agent needs to know its goal which describes desirable situations.
* Goal-based agents expand the capabilities of the model-based agent by having the "goal" information.
* They choose an action, so that they can achieve the goal.
* These agents may have to consider a long sequence of possible actions before deciding whether the goal is achieved or not. Such considerations of different scenario are called searching and planning, which makes an agent proactive.



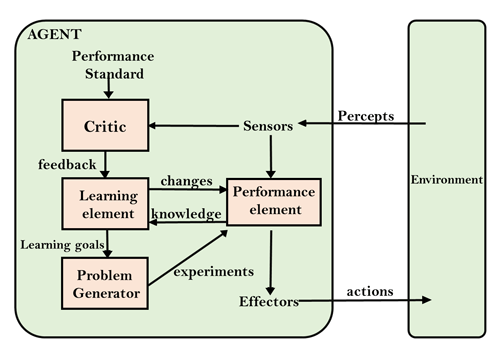
### **3.5. Utility-based agents**

* These agents are similar to the goal-based agent but provide an extra component of utility measurement which makes them different by providing a measure of success at a given state.
* Utility-based agent act based not only goals but also the best way to achieve the goal.
* The Utility-based agent is useful when there are multiple possible alternatives, and an agent has to choose in order to perform the best action.
* The utility function maps each state to a real number to check how efficiently each action achieves the goals.



### **3.6 Learning Agents**

* A learning agent in AI is the type of agent which can learn from its past experiences, or it has learning capabilities.
* It starts to act with basic knowledge and then able to act and adapt automatically through learning.
* A learning agent has mainly four conceptual components, which are:
  + **Learning element:** It is responsible for making improvements by learning from environment
  + **Critic:** Learning element takes feedback from critic which describes that how well the agent is doing with respect to a fixed performance standard.
  + **Performance element:** It is responsible for selecting external action
  + **Problem generator:** This component is responsible for suggesting actions that will lead to new and informative experiences.
* Hence, learning agents are able to learn, analyze performance, and look for new ways to improve the performance.



## **4. PROBLEM-SOLVING AGENTS**

The problem-solving agent perfoms precisely by defining problems and its several solutions.

According to psychology, “a problem-solving refers to a state where we wish to reach to a definite goal from a present state or condition.”

Therefore, a problem-solving agent is a **goal-driven agent** and focuses on satisfying the goal.

**Steps performed by Problem-solving agent**

* **Goal Formulation:** It is the first and simplest step in problem-solving. It organizes the steps/sequence required to formulate one goal out of multiple goals as well as actions to achieve that goal. Goal formulation is based on the current situation and the agent’s performance measure (discussed below).
* **Problem Formulation:** It is the most important step of problem-solving which decides what actions should be taken to achieve the formulated goal. There are following five components involved in problem formulation:
* **Initial State:** It is the starting state or initial step of the agent towards its goal.
* **Actions:** It is the description of the possible actions available to the agent.
* **Transition Model:** It describes what each action does.
* **Goal Test:** It determines if the given state is a goal state.
* **Path cost:** It assigns a numeric cost to each path that follows the goal. The problem-solving agent selects a cost function, which reflects its performance measure. Remember, **an optimal solution has the lowest path cost among all the solutions.**

**Note:** **Initial state, actions**, and **transition model** together define the **state-space** of the problem implicitly. State-space of a problem is a set of all states which can be reached from the initial state followed by any sequence of actions. The state-space forms a directed map or graph where nodes are the states, links between the nodes are actions, and the path is a sequence of states connected by the sequence of actions.

* **Search:** It identifies all the best possible sequence of actions to reach the goal state from the current state. It takes a problem as an input and returns solution as its output.
* **Solution:** It finds the best algorithm out of various algorithms, which may be proven as the best optimal solution.
* **Execution:** It executes the best optimal solution from the searching algorithms to reach the goal state from the current state.

# WEEK 6: SEARCHING FOR SOLUTIONS

## **1. SEARCHING FOR SOLUTIONS**

Artificial Intelligence is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.

- A search problem consists of:

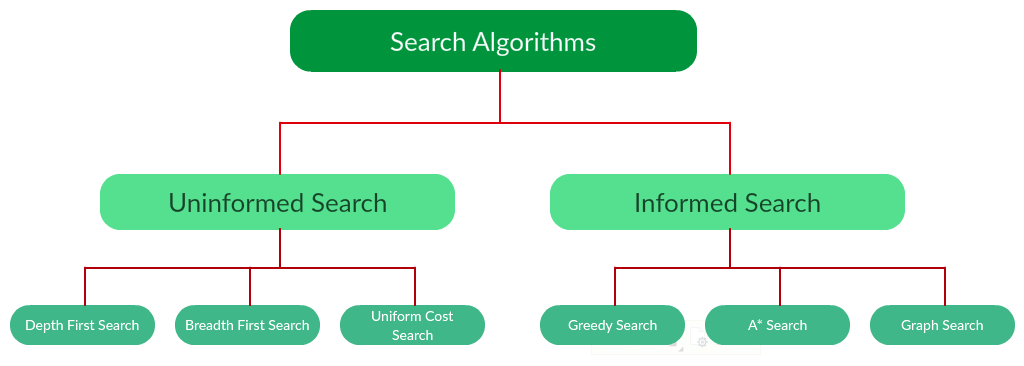
* **A State Space**. Set of all possible states where you can be.
* **A Start State**. The state from where the search begins.
* **A Goal Test**. A function that looks at the current state returns whether or not it is the goal state.

- The Solution to a search problem is a sequence of actions, called the plan that transforms the start state to the goal state.

- This plan is achieved through search algorithms.

- Types of search algorithms:

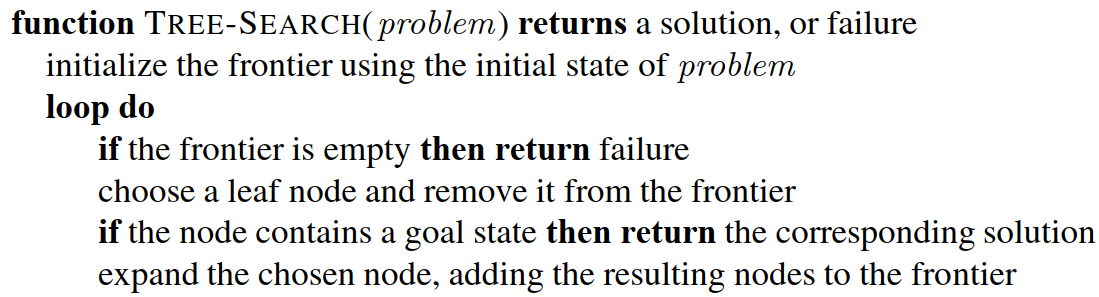
There are far too many powerful search algorithms out there to fit in a single article. Instead, this article will discuss six of the fundamental search algorithms, divided into two categories, as shown below.



## **2. TREE SEARCH AND GRAPH SEARCH**

### **2.1 Tree search**

**Search algorithms differ by the order in which they visit (reach) the states in the state graph following the edges between them.** For some algorithms, that order creates a tree superimposed over the state graph and whose root is the start state. We call that tree a search tree and will consider only the algorithms that grow it.

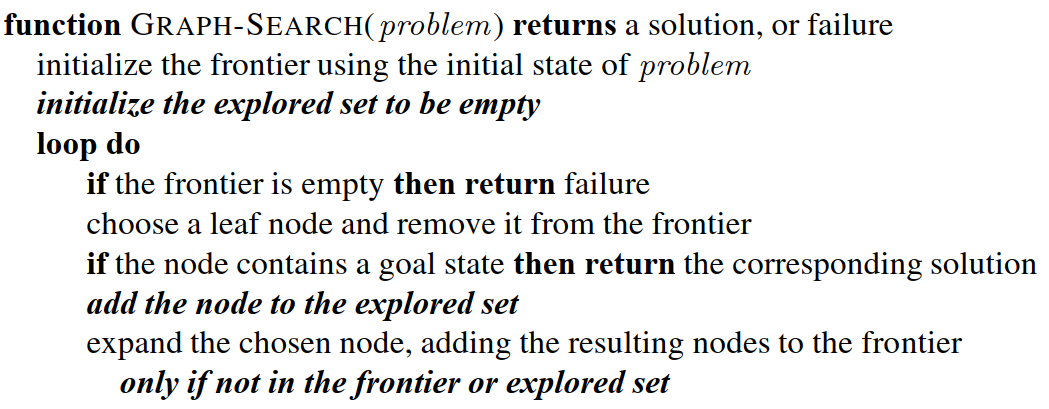


**2.2 Graph Search**

GS has one rule regarding the frontier:

**Don’t add a node if its state has already been expanded or a node pointing to the same state is already in the frontier.**

All the algorithms that conform to it belong to the class of graph-search methods. The generic pseudocode of GS is:



**3. Uninformed Search Algorithms:**

The search algorithms in this section have no additional information on the goal node other than the one provided in the problem definition. The plans to reach the goal state from the start state differ only by the order and/or length of actions. Uninformed search is also called Blind search.

The following uninformed search algorithms are discussed in this section.

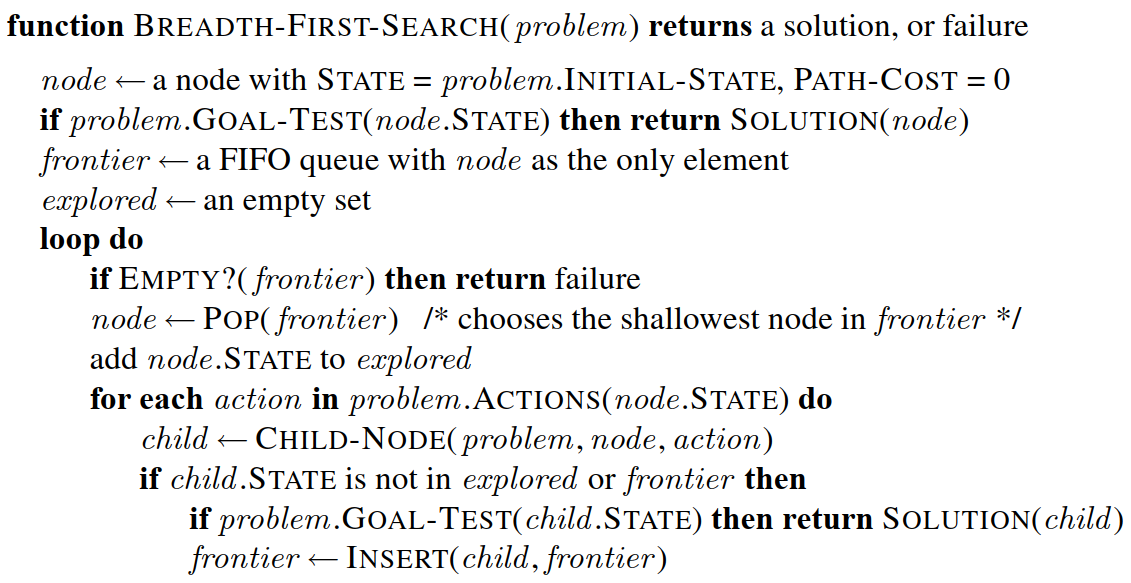
* Depth First Search
* Breadth First Search
* Uniform Cost Search

Each of these algorithms will have:

* A problem graph, containing the start node S and the goal node G.
* A strategy, describing the manner in which the graph will be traversed to get to G.
* A fringe, which is a data structure used to store all the possible states (nodes) that you can go from the current states.
* A tree, that results while traversing to the goal node.
* A solution plan, which the sequence of nodes from S to G.

**3.1 Breadth First Search:**

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a ‘search key’), and explores all of the neighbour nodes at the present depth prior to moving on to the nodes at the next depth level.



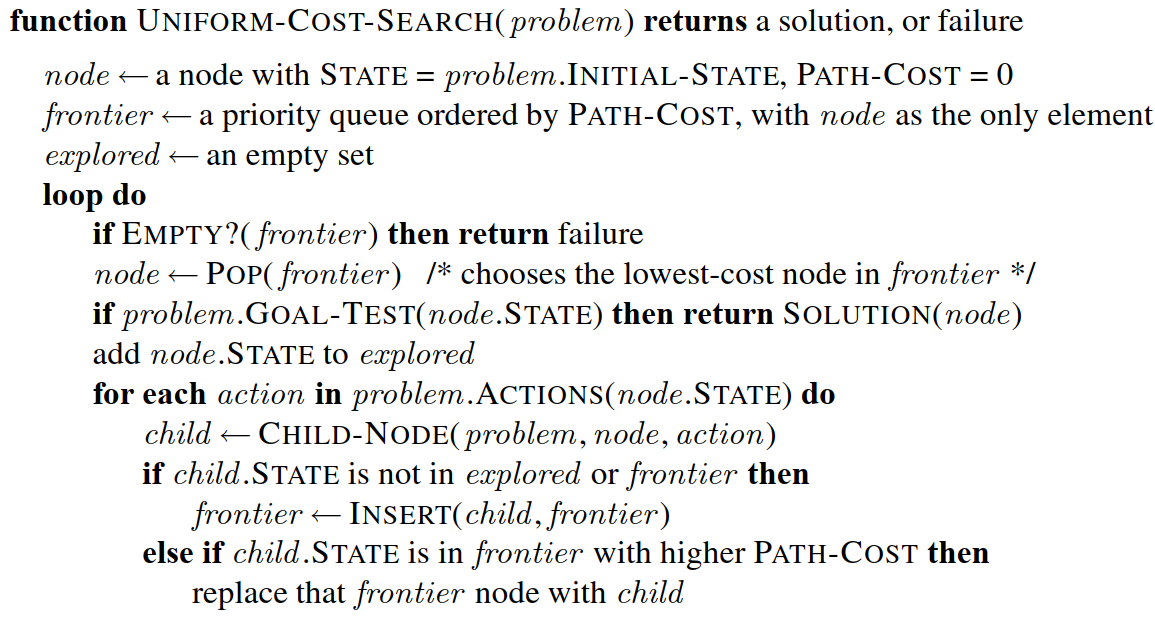
* s = the depth of the shallowest solution.
* n^i = number of nodes in level i .
* Time complexity: Equivalent to the number of nodes traversed in BFS until the shallowest solution. T(n) = 1 + n^2 + n^3 + ... + n^s = O(n^s)
* Space complexity: Equivalent to how large can the fringe get. S(n) = O(n^s)
* Completeness: BFS is complete, meaning for a given search tree, BFS will come up with a solution if it exists.
* Optimality: BFS is optimal as long as the costs of all edges are equal.

**3.2 Uniform-cost search algorithm**

UCS is different from BFS and DFS because here the costs come into play. In other words, traversing via different edges might not have the same cost. The goal is to find a path where the cumulative sum of costs is the least.

Cost of a node is defined as:

* cost(node) = cumulative cost of all nodes from root
* cost(root) = 0



* Then C / \varepsilon = effective depth
* Time complexity:   T(n) = O(n^C ^/ ^\varepsilon)
* Space complexity:   S(n) = O(n^C ^/ ^\varepsilon)

- Advantages:

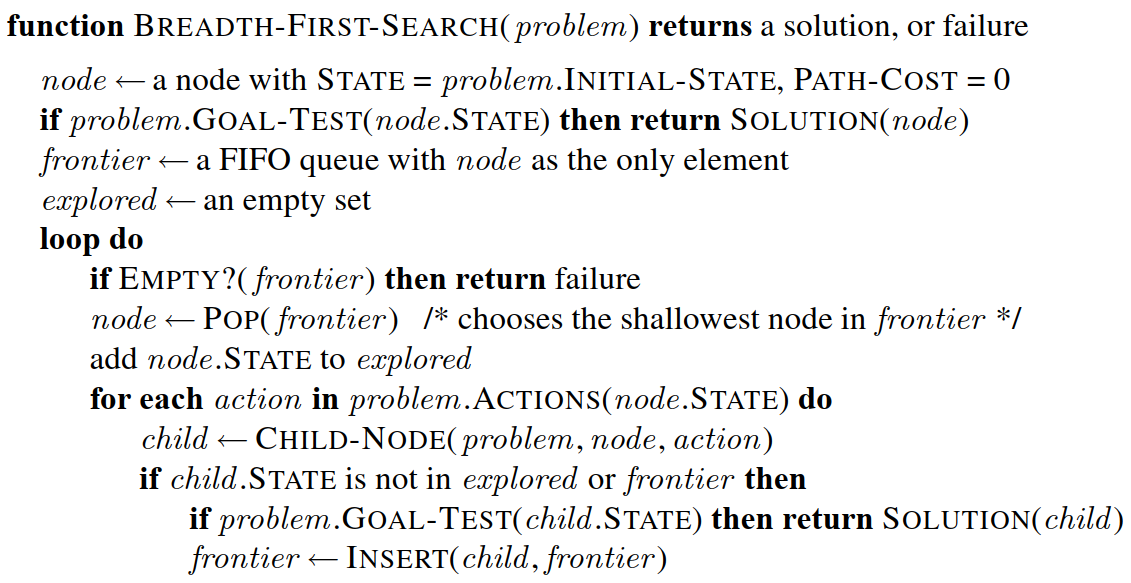
* UCS is complete only if states are finite and there should be no loop with zero weight.
* UCS is optimal only if there is no negative cost.

- Disadvantages:

* Explores options in every “direction”.
* No information on goal location.

**3.3 Depth First Search:**

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.



* d = the depth of the search tree = the number of levels of the search tree.
* n^i = number of nodes in level i .
* Time complexity: Equivalent to the number of nodes traversed in DFS. T(n) = 1 + n^2 + n^3 + ... + n^d = O(n^d)
* Space complexity: Equivalent to how large can the fringe get. S(n) = O(n \times d)
* Completeness: DFS is complete if the search tree is finite, meaning for a given finite search tree, DFS will come up with a solution if it exists.
* Optimality: DFS is not optimal, meaning the number of steps in reaching the solution, or the cost spent in reaching it is high.

# WEEK 7: INFORMED SEARCH ALGORITHMS

This section shows how an informed search strategy—one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than can an uninformed strategy

**Heuristics function**: Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Admissibility of the heuristic function is given as:

* h(n) <= h\*(n)

Here h(n) is heuristic cost, and h\*(n) is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

## **1. GREEDY BEST-FIRST SEARCH**

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

f(n)= g(n).

Were, h(n)= estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

**Best first search algorithm:**

* **Step 1:** Place the starting node into the OPEN list.
* **Step 2:** If the OPEN list is empty, Stop and return failure.
* **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
* **Step 4:** Expand the node n, and generate the successors of node n.
* **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
* **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
* **Step 7:** Return to Step 2.

**Advantages:**

* Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
* This algorithm is more efficient than BFS and DFS algorithms.

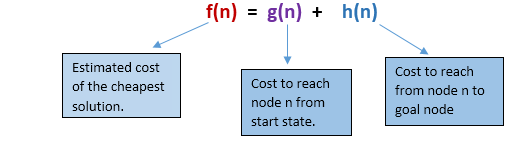
**Disadvantages:**

* It can behave as an unguided depth-first search in the worst case scenario.
* It can get stuck in a loop as DFS.
* This algorithm is not optimal.

## **2. A\* SEARCH ALGORITHM**

A\* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A\* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A\* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

In A\* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a **fitness number**.



**Algorithm of A\* search:**

* **Step1:** Place the starting node in the OPEN list.
* **Step 2:** Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
* **Step 3:** Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise
* **Step 4:** Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
* **Step 5:** Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
* **Step 6:** Return to **Step 2**.

**Advantages:**

* A\* search algorithm is the best algorithm than other search algorithms.
* A\* search algorithm is optimal and complete.
* This algorithm can solve very complex problems.

**Disadvantages:**

* It does not always produce the shortest path as it mostly based on heuristics and approximation.
* A\* search algorithm has some complexity issues.
* The main drawback of A\* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

## **3.** **HEURISTIC FUNCTIONS**

### **3.1 The effect of heuristic accuracy on performance**

One way to characterize the quality of a heuristic is the effective branching factor b∗. If the total number of nodes generated by A∗ for a particular problem is N and the solution depth is d, then b∗ is the branching factor that a uniform tree of depth d would have to have in order to contain N + 1 nodes. Thus,

**N + 1 = 1 + b∗ + + · · · +**

### **3.2. Generating admissible heuristics from relaxed problems**

A problem with fewer restrictions on the actions is called a relaxed problem. The state-space graph of the relaxed problem is a supergraph of the original state space because the removal of restrictions creates added edges in the graph.

Because the relaxed problem adds edges to the state space, any optimal solution in the original problem is, by definition, also a solution in the relaxed problem; but the relaxed problem may have better solutions if the added edges provide short cuts. Hence, the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem. Furthermore, because the derived heuristic is an exact cost for the relaxed problem, it must obey the triangle inequality and is therefore consistent

### **3.3. Generating admissible heuristics from subproblems: Pattern databases**

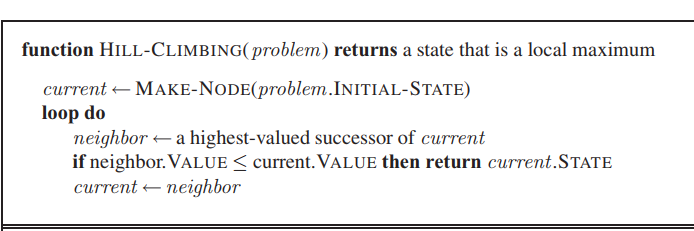
Admissible heuristics can also be derived from the solution cost of a subproblem of a given problem. The idea behind pattern databases is to store these exact solution costs for every possible subproblem instance

# WEEK 8 : SEARCHING IN MORE COMPLEX ENVIRONMENTS

## **1. LOCAL SEARCH**

Local search algorithms operate by searching from a start state to neighboring states,

without keeping track of the paths, nor the set of states that have been reached. That means they are not systematic—they might never explore a portion of the search space where a solution actually resides. However, they have two key advantages: (1) they use very little memory; and (2) they can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.



**1.1 Hill Climbing Search**

Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.

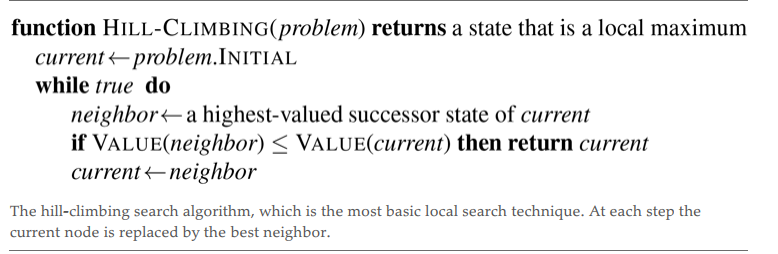
Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.

It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.

A node of hill climbing algorithm has two components which are state and value.

Hill Climbing is mostly used when a good heuristic is available.

In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.



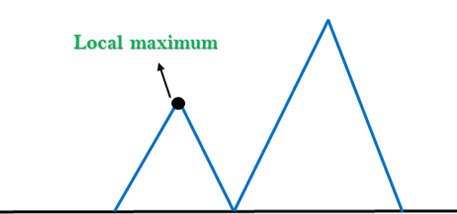
Features of Hill Climbing

Variant of generate and test algorithm: It is a variant of generating and test algorithm. The generate and test algorithm is as follows :

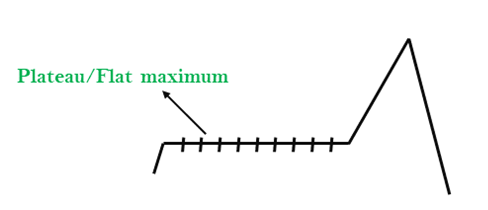
* Generate possible solutions.
* Test to see if this is the expected solution.
* If the solution has been found quit else go to step 1.

Problems in Hill Climbing Algorithm:

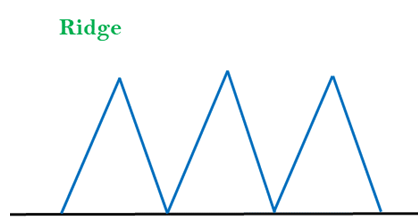
* **Local Maximum**: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.
* Solution: Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.



* **Plateau:** A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.
* **Solution:** The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.



* **Ridges:** A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.
* **Solution:** With the use of bidirectional search, or by moving in different directions, we can improve this problem.



## **2. LOCAL SEARCH OTHER**

### **2.1 Local beam search**

A heuristic search algorithm that examines a graph by extending the most promising node in a limited set is known as beam search.

Beam search is a heuristic search technique that always expands the W number of the best nodes at each level. It progresses level by level and moves downwards only from the best W nodes at each level. Beam Search uses breadth-first search to build its search tree. Beam Search constructs its search tree using breadth-first search. It generates all the successors of the current level’s state at each level of the tree. However, at each level, it only evaluates a W number of states. Other nodes are not taken into account.

The heuristic cost associated with the node is used to choose the best nodes. The width of the beam search is denoted by W. If B is the branching factor, at every depth, there will always be W × B nodes under consideration, but only W will be chosen. More states are trimmed when the beam width is reduced.

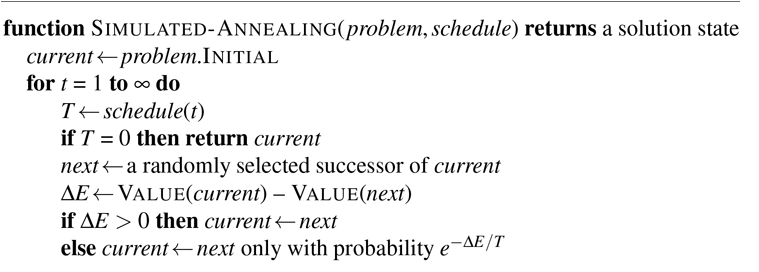
When W = 1, the search becomes a hill-climbing search in which the best node is always chosen from the successor nodes. No states are pruned if the beam width is unlimited, and the beam search is identified as a breadth-first search.

The beamwidth bounds the amount of memory needed to complete the search, but it comes at the cost of completeness and optimality (possibly that it will not find the best solution). The reason for this danger is that the desired state could have been pruned.

**2.2 Simulated annealing**

A hill-climbing algorithm which never makes a move towards a lower value guaranteed to be incomplete because it can get stuck on a local maximum. And if algorithm applies a random walk, by moving a successor, then it may complete but not efficient. Simulated Annealing is an algorithm which yields both efficiency and completeness.

In mechanical term Annealing is a process of hardening a metal or glass to a high temperature then cooling gradually, so this allows the metal to reach a low-energy crystalline state. The same process is used in simulated annealing in which the algorithm picks a random move, instead of picking the best move. If the random move improves the state, then it follows the same path. Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path.



# WEEK 9: SEARCHING IN NONDETERMINISTIC ENVIRONMENTS

## **1 AND–OR SEARCH TREES**

Given an initial problem P0 and set of problem solving methods of the form:

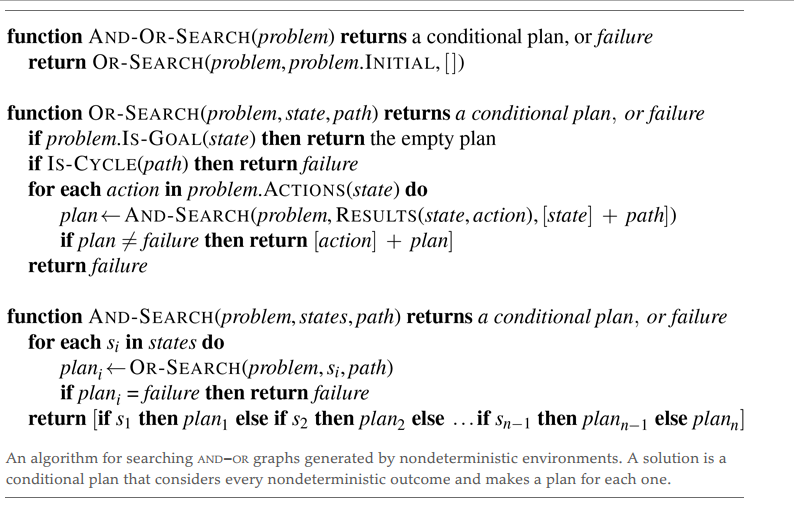
P if P1 and … and Pn

the associated and-or tree is a set of labelled nodes such that:

1. The root of the tree is a node labelled by P0.
2. For every node N labelled by a problem or sub-problem P and for every method of the form P if P1 and … and Pn, there exists a set of children nodes N1, …, Nn of the node N, such that each node Ni is labelled by Pi. The nodes are conjoined by an arc, to distinguish them from children of N that might be associated with other methods.

A node N, labelled by a problem P, is a success node if there is a method of the form P if nothing (i.e., P is a "fact"). The node is a failure node if there is no method for solving P.

If all of the children of a node N, conjoined by the same arc, are success nodes, then the node N is also a success node. Otherwise the node is a failure node.



## **2. SEARCHING IN PARTIALLY OBSERVABLE ENVIROMENTS**

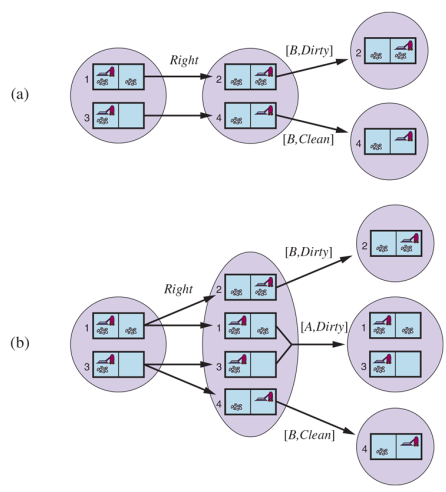
For a partially observable problem, the problem specification will specify a PERCEPT(s)  
function that returns the percept received by the agent in a given state. If sensing is  
nondeterministic, then we can use a PERCEPTS function that returns a set of possible percepts. For fully observable problems, PERCEPT(s) = s for every state s , and for sensorless problems  
PERCEPT(s) = null

* The **prediction** stage computes the belief state resulting from the action, ,  
  exactly as we did with sensorless problems. To emphasize that this is a prediction, weuse the notation b = RESULT(b, a), where the “hat” over the means “estimated,” and we also use PREDICT(b, a) as a synonym for RESULT(b, a)
* The **possible percepts** stage computes the set of percepts that could be observed in thepredicted belief state (using the letter for observation)

POSSIBLE-PERCEPTS(ˆb) = {o : o = PERCEPT(s) and s ∈ ˆb} .

* The **update** stage computes, for each possible percept, the belief state that would result from the percept. The updated belief state is the set of states in that could have produced the percept:

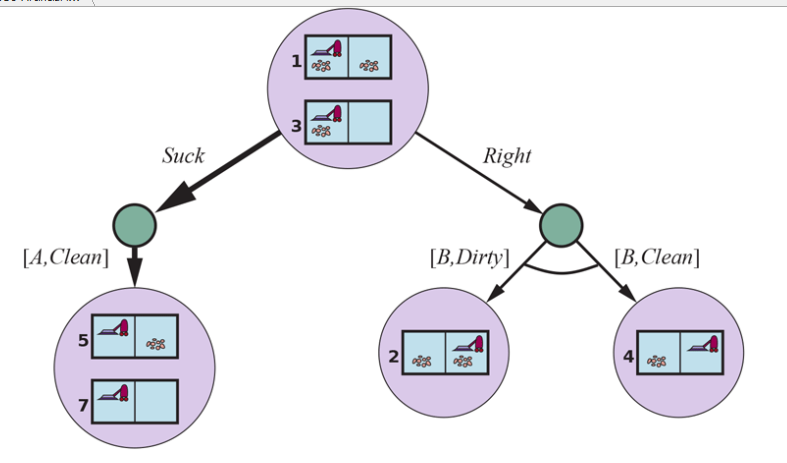
bo = UPDATE(ˆb,o) = {s : o = PERCEPT(s) and s ∈ ˆb} .



### **2.1 Solving partially observable problems**

The preceding section showed how to derive the RESULTS function for a nondeterministicbelief-state problem from an underlying physical problem, given the PERCEPT function. With this formulation, the AND–OR search algorithm of Figure 4.11 can be applied directly to derive a solution The solution is the conditional plan

[Suck,Right, if Bstate = {6} then Suck else [ ]] .

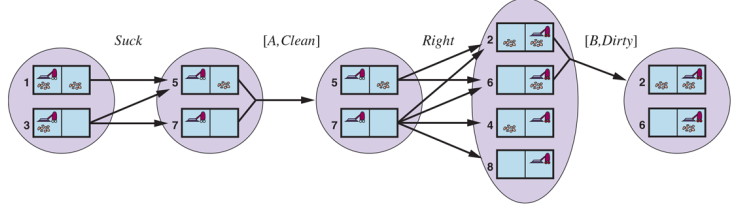
****

As in the case of standard search algorithms applied to sensorless problems, the AND–OR search algorithm treats belief states as black boxes, just like any other states. One can  
improve on this by checking for previously generated belief states that are subsets or  
supersets of the current state, just as for sensorless problems. One can also derive  
incremental search algorithms, analogous to those described for sensorless problems, that  
provide substantial speedups over the black-box approach

### **2.2 An agent for partially observable environments**

An agent for partially observable environments formulates a problem, calls a search  
algorithm (such as AND-OR-SEARCH) to solve it, and executes the solution. There are two main differences between this agent and the one for fully observable deterministic  
environments. First, the solution will be a conditional plan rather than a sequence; to  
execute an if–then–else expression, the agent will need to test the condition and execute the appropriate branch of the conditional. Second, the agent will need to maintain its belief state as it performs actions and receives percepts. Given an initial belief state , an action , and a percept , the new belief state is





# WEEK 10: SEARCHING WITH NO TRANSITION MODEL – ONLINE SEARCH

## **1. ONLINE SEARCH**

### **1.1 Online search problems**

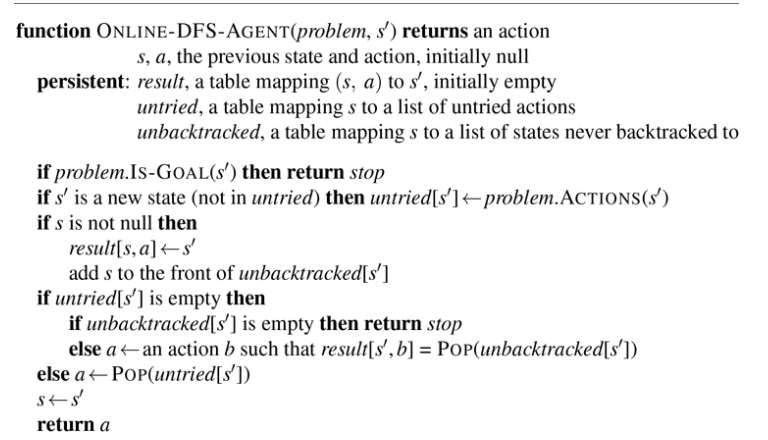
An online search problem is solved by interleaving computation, sensing, and acting. We’ll start by assuming a deterministic and fully observable environment and stipulate that the agent knows only the following:

* ACTIONS(s), the legal actions in state ;
* c(s,a,s′), the cost of applying action in state to arrive at state . Note that this cannot
* be used until the agent knows that is the outcome.
* Is-GOAL(s), the goal test

### **2 Online search agents**

After each action, an online agent in an observable environment receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. The updated map is then used to plan where to go next. This interleaving of planning and action means that online search algorithms are quite different from the offline search algorithms we have seen previously: offline algorithms explore their model of the state space, while online algorithms explore the real world.

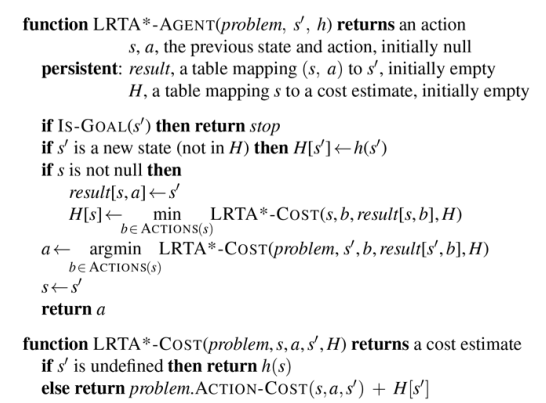
An online algorithm, on the other hand, can discover successors only for a state that it physically occupies. To avoid traveling all the way to a distant state to expand the next node, it seems better to expand nodes in a local order. Depth-first search has exactly this property because (except when the algorithm is backtracking) the next node expanded is a child of the previous node expanded.



### **3 Online local search**

Like depth-first search, hill-climbing search has the property of locality in its node expansions. In fact, because it keeps just one current state in memory, hill-climbing search is already an online search algorithm! Unfortunately, the basic algorithm is not very good for exploration because it leaves the agent sitting at local maxima with nowhere to go. Moreover, random restarts cannot be used, because the agent cannot teleport itself to a new start state.

An agent implementing this scheme, which is called learning real-time A\* (LRTA\*), is shown in Figure. Like ONLINE-DFS-AGENT, it builds a map of the environment in the result table. It updates the cost estimate for the state it has just left and then chooses the “apparently best” move according to its current cost estimates. One important detail is that actions that have not yet been tried in a state are always assumed to lead immediately to the goal with the least possible cost, namely . This optimism under uncertainty encourages the agent to explore new, possibly promising paths.



## **2. CONSTRAINT SATISFACTOIN PROBLEMS**

### **2.1 Definition**

A constraint satisfaction problem consists of three components, X,D, and C:

* X is a set of variables, {X1,... ,Xn}.
* D is a set of domains, {D1,... ,Dn}, one for each variable.
* C is a set of constraints that specify allowable combinations of values.

Each domain Di consists of a set of allowable values, {v1,... ,vk} for variable Xi. Each constraint Ci consists of a pair <scope, rel>, where scope is a tuple of variables that articipate in the constraint and rel is a relation that defines the values that those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation.

To solve a CSP, we need to define a state space and the notion of a solution. Each  
state in a CSP is defined by an **assignment** of values to some or all of the variables, {Xi = vi,Xj = vj,...}. An assignment that does not violate any constraints is called a **consistent** or legal assignment. A **complete assignment** is one in which every variable is assigned, and SOLUTION a **solution** to a CSP is a consistent, complete assignment. A **partial assignment** is one that assigns values to only some of the variables.

### **2.2 Constraint propagation: inference in CSPS**

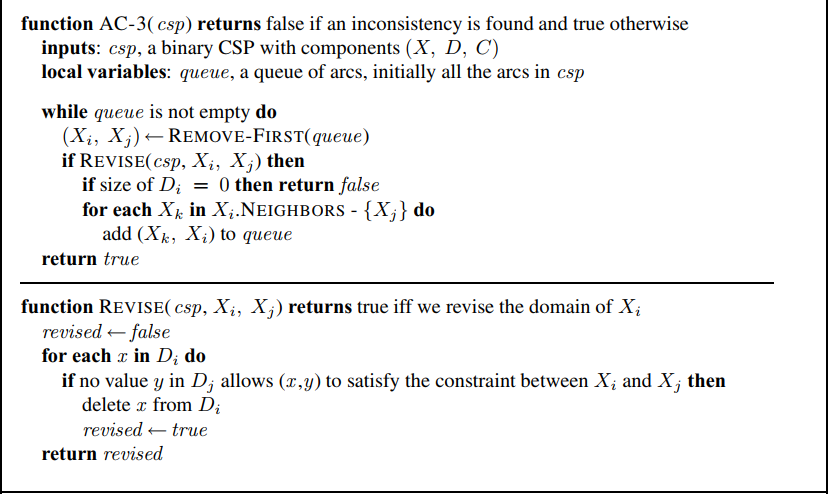
* **Node consistency**

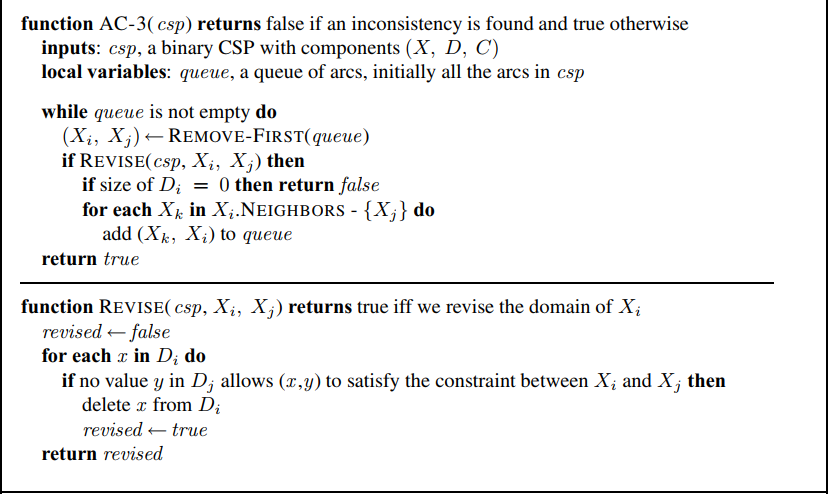
A single variable (corresponding to a node in the CSP network) is node-consistent if all

the values in the variable’s domain satisfy the variable’s unary constraints

* **Arc consistency**

A variable in a CSP is arc-consistent if every value in its domain satisfies the variable’s binary constraints. More formally, Xi is arc-consistent with respect to another variable Xj if for every value in the current domain Di there is some value in the domain Dj that satisfies the binary constraint on the arc (Xi,Xj). A network is arc-consistent if every variable is arc consistent with every other variable.





* **Path consistency**

Arc consistency tightens down the domains (unary constraints) using the arcs (binary constraints). To make progress on problems like map coloring, we need a stronger notion of consistency. Path consistency tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables.

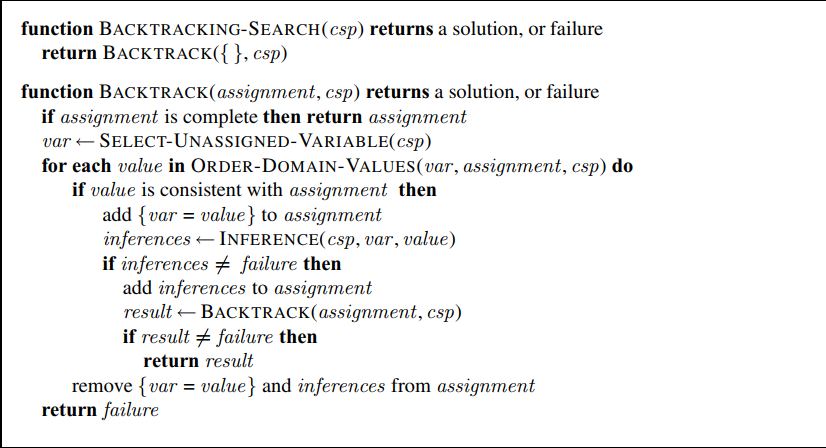
* ***K*-consistency**

Stronger forms of propagation can be defined with the notion of k-consistency. A CSP is k-consistent if, for any set of k − 1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable. 1-consistency says that, given the empty set, we can make any set of one variable consistent: this is what we called node consistency. 2-consistency is the same as arc consistency. For binary constraint networks, 3-consistency is the same as path consistency

# WEEK 11: BACKTRACKING SEARCH FOR CSPS

Our seemingly reasonable but naive formulation ignores crucial property common to  
COMMUTATIVITY all CSPs: **commutativity**. A problem is commutative if the order of application of any given set of actions has no effect on the outcome. CSPs are commutative because when assigning values to variables, we reach the same partial assignment regardless of order. Therefore, we need only consider a *single* variable at each node in the search tree

The term backtracking search is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.



## **1 Variable and value ordering**

The backtracking algorithm contains the line

var ← SELECT-UNASSIGNED-VARIABLE(csp) .

The simplest strategy for SELECT-UNASSIGNED-VARIABLE is to choose the next unassigned variable in order, {X1,X2,...}. This static variable ordering seldom results in the most efficient search. In fact, after SA is assigned, the choices for Q, NSW , and V are all forced. This intuitive idea—choosing the variable with the fewest “legal” values—is called the **remaining-values** (MRV) heuristic

In fact, after SA is assigned, the choices for Q, NSW , and V are all forced. This  
intuitive idea—choosing the variable with the fewest “legal” values—is called the **minimum**MINIMUM- REMAINING-VALUES **remaining-values** (MRV) heuristic

## **2 Interleaving search and inference**

We saw how AC-3 can reduce the domains of variables before we begin the search. But inference can be even more powerful during the course of a search: every time we make a choice of a value for a variable, we have a brand-new opportunity to infer new domain reductions on the neighboring variables.

One of the simplest forms of inference is called forward checking. Whenever a variable is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable that is connected to by a constraint, delete from ’s domain any value that is inconsistent with the value chosen for X.

The algorithm called MAC (for Maintaining Arc Consistency) detects inconsistencies like this. After a variable is assigned a value, the INFERENCE procedure calls AC-3, but instead of a queue of all arcs in the CSP, we start with only the arcs for all that are unassigned variables that are neighbors of . From there, AC-3 does constraint propagation in the usual way, and if any variable has its domain reduced to the empty set, the call to AC-3 fails and we know to backtrack immediately. We can see that MAC is strictly more powerful than forward checking because forward checking does the same thing as MAC on the initial arcs in MAC’s queue; but unlike MAC, forward checking does not recursively propagate constraints when changes are made to the domains of variables

## **3 Intelligent backtracking: Looking backward**

The BACKTRACKING-SEARCH algorithm has a very simple policy for what to do

when a branch of the search fails: back up to the preceding variable and try a different value

for it. This is called chronological backtracking because the most recent decision point is

revisited. In this subsection, we consider better possibilities.

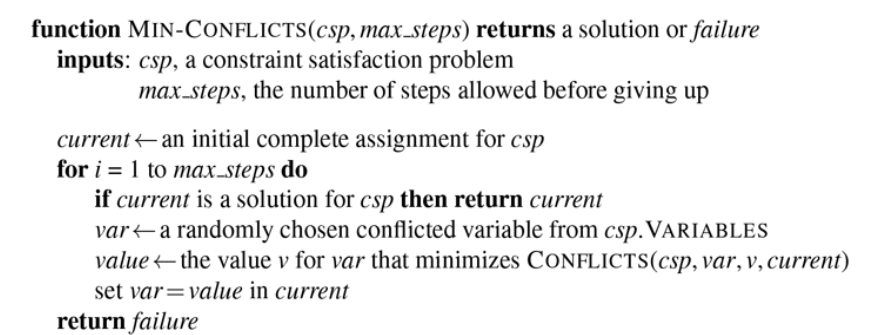
## **4 Constraint learning**

When we reach a contradiction, back jumping can tell us how far to back up, so we don’t waste time changing variables that won’t fix the problem. But we would also like to avoid running into the same problem again. When the search arrives at a contradiction, we know that some subset of the conflict set is responsible for the problem. Constraint learning is the idea of finding a minimum set of variables from the conflict set that causes the problem. This set of variables, along with their corresponding values, is called a no-good. We then record the no-good, either by adding a new constraint to the CSP to forbid this combination of assignments or by keeping a separate cache of no-goods

# WEEK 12 SOLVE CSPS

## **1. SOLVE CSPS USING LOCAL SEARCH**

**1.1 Min-conflicts**



The MIN-CONFLICTS local search algorithm for CSPs. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn

## **2. Solve CSPs using Constraint Graphs**

### **2.1 Tree-structured CSP**

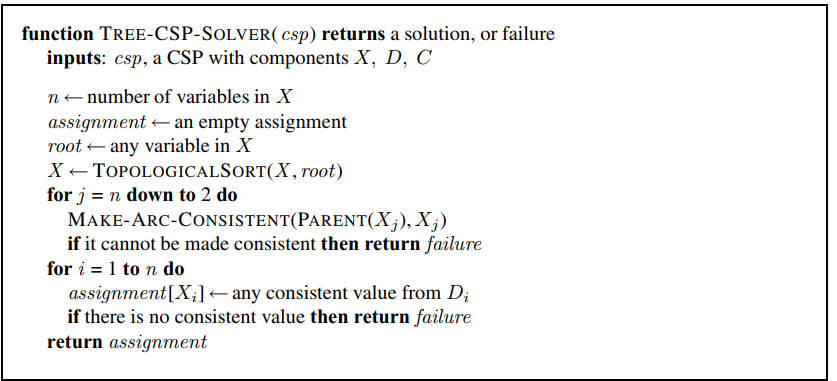
The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

Main steps to solve a tree-structed CSP:

1. Pick a variable to be the root. Sort variables such that each variable appears after its parent

2. Enforce arc consistency on sorted variables

3. Choose values for variables from consistent domains.



Now, any solution for the CSP after SA and its constraints are removed will be consistent with the value chosen for SA. Therefore, we can solve the remaining tree with the algorithm given above and thus solve the whole problem. Of course, in the general case (as opposed to map coloring), the value chosen for SA could be the wrong one, so we would need to try each possible value. The general algorithm is as follows:

1. Choose a subset S of the CSP’s variables such that the constraint graph becomes a treeafter removal of S. S is called a **cycle cutset**.  
2. For each possible assignment to the variables in S that satisfies all constraints on S,

(a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S, and  
(b) If the remaining CSP has a solution, return it together with the assignment for S.

A tree decomposition must satisfy the following three requirements:

• Every variable in the original problem appears in at least one of the subproblems.

• If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.

• If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.

# WEEK 13 INTRODUCE REINFORCEMENT LEARNING

## **1. INTRODUCTION**

### **1.1 Supervised learning**

You may be familiar with the notion of supervised learning, which is the most studied and well-known machine learning problem. Its basic question is, how do you automatically build a function that maps some input into some output when given a set of example pairs? It sounds simple in those terms, but the problem includes many tricky questions that computers have only recently started to address with some success. There are lots of examples of supervised learning problems, including the following:

• Text classification: Is this email message spam or not?

• Image classification and object location: Does this image contain a picture of a cat, dog, or something else?

• Regression problems: Given the information from weather sensors, what will be the weather tomorrow?

• Sentiment analysis: What is the customer satisfaction level of this review?

### **1.2 Unsupervised learning**

The main objective is to learn some hidden structure of the dataset at hand. One common example of such an approach to learning is the clustering of data. This happens when our algorithm tries to combine data items into a set of clusters, which can reveal relationships in data. For instance, you might want to find similar images or clients with common behaviors.

Another unsupervised learning method that is becoming more and more popular is generative adversarial networks (GANs). When we have two competing neural networks, the first network is trying to generate fake data to fool the second network, while the second network is trying to discriminate artificially generated data from data sampled from our dataset. Over time, both networks become more and more skillful in their tasks by capturing subtle specific patterns in the dataset.

### **1.3 Reinforcement learning**

RL is the third camp and lies somewhere in between full supervision and a complete lack of predefined labels. On the one hand, it uses many well-established methods of supervised learning, such as deep neural networks for function approximation, stochastic gradient descent, and backpropagation, to learn data representation. On the other hand, it usually applies them in a different way

## **2 CROSS-ENTROPY METHOD**

The cross-entropy (CE) method is a Monte Carlo method for importance sampling and optimization. It is applicable to both combinatorial and continuous problems, with either a static or noisy objective.

The method approximates the optimal importance sampling estimator by repeating two phases:

* Draw a sample from a probability distribution.
* Minimize the cross-entropy between this distribution and a target distribution to produce a better sample in the next iteration.

Reuven Rubinstein developed the method in the context of rare event simulation, where tiny probabilities must be estimated, for example in network reliability analysis, queueing models, or performance analysis of telecommunication systems. The method has also been applied to the traveling salesman, quadratic assignment, DNA sequence alignment, max-cut and buffer allocation problems.