

Artificial Intelligence I

Ibrahim Nasser

November 22, 2025

Contents

1	Recap	2
1.1	Set Theory	2
1.2	Relations	2
1.3	Functions	3
1.4	Mathematical Structures	4
1.5	Formal Languages and Grammars	5
1.6	Graphs and Trees	7
2	Introduction	9
3	General Problem Solving	11

1 Recap

1.1 Set Theory

Def 1.1. Set: A collection of different things (called **elements** or **members** of the set). We can represent **sets** by listing the **elements** within curly brackets, e.g. $\{a, b, c\}$ or by describing the **elements** via a property, e.g. $\{x \mid x \bmod 2 = 0\}$ (the **set** of even numbers). We can state **element-hood** ($a \in S$) or not ($b \notin S$)

Def 1.2. Set Equality: $A \equiv B := \forall x. x \in A \Leftrightarrow x \in B$

Def 1.3. Subset: $A \subseteq B := \forall x. x \in A \Rightarrow x \in B$

Def 1.4. Proper Subset: $A \subset B := (A \subseteq B) \wedge (A \neq B)$

Def 1.5. Super Set: $A \supseteq B := \forall x. x \in B \Rightarrow x \in A$

Def 1.6. Proper Superset: $A \supset B := (A \supseteq B) \wedge (A \neq B)$

Def 1.7. Set Union: $A \cup B := \{x \mid x \in A \vee x \in B\}$

Def 1.8. Set Intersection: $A \cap B := \{x \mid x \in A \wedge x \in B\}$

Def 1.9. Disjoint: Two **sets** A, B are **disjoint** iff $A \cap B = \emptyset$

Def 1.10. Set Difference: $A \setminus B := \{x \mid x \in A \wedge x \notin B\}$

Def 1.11. Power Set: $\mathcal{P}(A) := \{S \mid S \subseteq A\}$ (the set of all subsets)

Def 1.12. Cartesian Product: $A \times B := \{(a, b) \mid a \in A \wedge b \in B\}$, (a, b) is a pair

Def 1.13. Family: A **set** of **sets**

Def 1.14. Topology: A **family** of open **sets**

Def 1.15. Indexing Function: Let I and X be **sets**. A function $f : I \rightarrow X$ is called an **indexing function**. The set I is called the **index set**. For each $i \in I$, we define $x_i := f(i)$, here i is called the **index** (or **parameter**) of x_i .

Def 1.16. Indexed Family: Given an **indexing function** $f : I \rightarrow X$, the **set** of values $f(I) = \{f(i) : i \in I\}$ is called an **indexed family**. It is often written as $(x_i)_{i \in I}$, $\langle x_i \rangle_{i \in I}$, or $\{x_i\}_{i \in I}$.

Def 1.17. Union over a Family: Let $(X_i)_{i \in I}$ be an **indexed family**, then $\bigcup_{i \in I} X_i := \{x \mid \exists i \in I. x \in X_i\}$

Def 1.18. Intersection over a Family: Let $(X_i)_{i \in I}$ be an **indexed family**, then $\bigcap_{i \in I} X_i := \{x \mid \forall i \in I. x \in X_i\}$

Def 1.19. n -fold Cartesian Product: $\prod_{i=1}^n X_i := X_1 \times \cdots \times X_n := \{\langle x_1, \dots, x_n \rangle \mid \forall i. 1 \leq i \leq n \Rightarrow x_i \in X_i\}$ where $\langle x_1, \dots, x_n \rangle$ is called an n -tuple.

Def 1.20. n -dim Cartesian Space: $X^n := \{\langle x_1, \dots, x_n \rangle \mid 1 \leq i \leq n \Rightarrow x_i \in X\}$ where $\langle x_1, \dots, x_n \rangle$ is called a vector.

Def 1.21. Size of a Set: The size $\Gamma(A)$ (or $|A|$) of a **set** A is the number of **elements** in A .

1.2 Relations

Def 1.22. Relation: $R \subseteq A \times B$ is a (binary) relation between A and B

Def 1.23. Relation on: a **relation** $R \subseteq A \times B$ where $A = B$ is called a **relation on** A

Def 1.24. Total: A **relation** $R \subseteq A \times B$ is called **total** (*left total*) iff $\forall x \in A. \exists y \in B. (x, y) \in R$

Def 1.25. Converse Relation: $R^{-1} \subseteq B \times A := \{(y, x) \mid (x, y) \in R\}$ is the **converse relation** of R

Def 1.26. Relation Composition: The composition of $R \subseteq A \times B$ and $S \subseteq B \times C$ is defined as $R \circ S := \{(a, c) \in A \times C \mid \exists b \in B. (a, b) \in R \wedge (b, c) \in S\}$

A **relation on** A : $R \subseteq A \times A$ is called:

- **reflexive** on A , iff $\forall a \in A. (a, a) \in R$
- **irreflexive** on A , iff $\forall a \in A. (a, a) \notin R$
- **symmetric** on A , iff $\forall a, b \in A. (a, b) \in R \Rightarrow (b, a) \in R$
- **asymmetric** on A , iff $\forall a, b \in A. (a, b) \in R \Rightarrow (b, a) \notin R$
- **antisymmetric** on A , iff $\forall a, b \in A. (a, b) \in R \wedge (b, a) \in R \Rightarrow a = b$

- **transitive** on A , iff $\forall a, b, c \in A. (a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$
- **equivalence relation** on A , iff R is **reflexive symmetric**, and **transitive**

Def 1.27. Equality Relation: The equality relation is an **equivalence relation** on any set.

Def 1.28. Divides Relation: The **divides relation** on the integers is defined as $| \subseteq \mathbb{Z} \times \mathbb{Z}$, where $| = \{(a, b) \in \mathbb{Z} \times \mathbb{Z} \mid \exists k \in \mathbb{Z} : b = a \cdot k\}$. We write $a | b$ to denote $(a, b) \in |$ (read as "a divides b").

Def 1.29. Congruence Modulo: For a fixed $n \in \mathbb{N}$ with $n \geq 1$, the **congruence modulo n relation** on the integers is defined as

$$\equiv_n \subseteq \mathbb{Z} \times \mathbb{Z}, \quad \equiv_n = \{(a, b) \in \mathbb{Z} \times \mathbb{Z} \mid n | (a - b)\}.$$

We write $a \equiv b \pmod{n}$ to denote $(a, b) \in \equiv_n$ (read as "a is congruent to b modulo n").

Def 1.30. Partial Ordering (\preceq): A relation $R \subseteq A \times A$ is called a (non-strict) **partial ordering** on A , iff R is **reflexive**, **antisymmetric**, and **transitive** on A .

Def 1.31. Strict Partial Ordering (\prec): A relation $R \subseteq A \times A$ is called a **strict partial ordering** on A , iff R is **irreflexive**, and **transitive** on A .

Def 1.32. Comparable: In a **non-strict partial ordering**, two elements a, b are comparable if either $a \preceq b$ or $b \preceq a$. If neither holds, they are incomparable.

Def 1.33. Linear Order: A **partial ordering** is called linear (or total) on A , iff all **elements** in A are **comparable**, i.e. if $(x, y) \in R$ or $(y, x) \in R$ for all $x, y \in A$. For example, the \leq **relation** is a **linear order** on \mathbb{N} . However, the "divides" ($|$) **relation** is a **non-strict partial ordering** on \mathbb{N} that is not **linear** (e.g., neither $2 | 3$ nor $3 | 2$).

1.3 Functions

Def 1.34. Partial function: A relation $R \subseteq X \times Y$ is a **partial function** from X to Y ($f : X \rightharpoonup Y$) iff $\forall x \in X, \forall y_1, y_2 \in Y. ((x, y_1) \in R \wedge (x, y_2) \in R \Rightarrow y_1 = y_2)$ (i.e. there is at most one $y \in Y$ with $(x, y) \in f$)

Def 1.35. dom, codom: We call X the domain ($\text{dom}(f)$), and Y the codomain ($\text{codom}(f)$)

Def 1.36. Function Application: Instead of writing $(x, y) \in f$ we write $f(x) = y$

Def 1.37. Undefined: we call a **partial function** $f : X \rightharpoonup Y$ undefined at $x \in X$, iff $\forall y \in Y. (x, y) \notin f$ ($f(x) = \perp$).

Def 1.38. Function: A **partial function** $f : X \rightharpoonup Y$ is called a **function** (or **total function**) from X to Y iff it is a **total relation**. ($\forall x \in X. \exists^1 y \in Y : (x, y) \in f$)

Note

A **partial function** guarantees uniqueness, but not existence. So each x has at most one y (some might have none). A **total function** guarantees both uniqueness and existence (at most and at least \rightarrow exactly one)

A **function** $f : X \rightarrow Y$ is called

- **injective** iff $\forall x_1, x_2 \in X. f(x_1) = f(x_2) \Rightarrow x_1 = x_2$
- **surjective** iff $\forall y \in Y. \exists x \in X. f(x) = y$
- **bijective** iff f is **injective** and **surjective**

Def 1.39. Function Composition: If $f : A \rightarrow B$ and $g : B \rightarrow C$ are **functions**, then we call $g \circ f : A \rightarrow C; x \mapsto g(f(x))$ the composition of g and f (read g after f).

Def 1.40. Image and Preimage: Let $f : X \rightarrow Y$ be a **function**, $X' \subseteq X$ and $Y' \subseteq Y$, then we call

- $f(X') := \{y \in Y \mid \exists x \in X'. (x, y) \in f\}$ the **image** of X' under f ,
- $\text{Im}(f) := f(X)$ the **image** of f , and
- $f^{-1}(Y') := \{x \in X \mid \exists y \in Y'. (x, y) \in f\}$ the **preimage** of Y' under f .

Def 1.41. Cardinality: We say that a set A is **finite** and has **cardinality** $\Gamma(A) \in \mathbb{N}$, iff there is a **bijective function** $f : A \rightarrow \{n \in \mathbb{N} \mid n < \Gamma(A)\}$.

Def 1.42. Countably Infinite: We say that a set A is countably infinite, iff there is a **bijective function** $f : A \rightarrow \mathbb{N}$.

Def 1.43. Countable: A set A is called **countable**, iff it is **finite** or **countably infinite**.

Def 1.44. Curried Function Type: Let X, Y, Z be sets. An *uncurried function* is written as

$$f : X \times Y \rightarrow Z, \quad f(x, y) = E(x, y).$$

The same **function** can equivalently be written in *curried* form as

$$f : X \rightarrow Y \rightarrow Z, \quad f(x)(y) = E(x, y).$$

Thus in the curried notation the **function** takes one argument at a time: for $x \in X$, the value $f(x)$ is itself a **function** $Y \rightarrow Z$, and applying it to $y \in Y$ yields $f(x)(y) \in Z$.

Def 1.45. Invertible: Let A and B be sets and $f : A \rightarrow B$ a **function**, then f is called **invertible**, iff there is a **function** $g : B \rightarrow A$, such that $f \circ g = \text{Id}_B$. g is called the **inverse function** (or just **inverse**) of f and is written as f^{-1} .

1.4 Mathematical Structures

Def 1.46. Formulae: A mathematical formula can be a **mathematical statement** (clause) which can be true or false (e.g. $x > 5, 3 + 5 = 7$), or a **mathematical object** (e.g. $3, n, x^2 + y^2 + z^2, \int_1^0 x^{3/2} dx$)

Def 1.47. Mathematical Structure: A mathematical structure combines multiple **mathematical objects** (the components) into a new **object**. The components usually have names by which they can be referenced. Given a definition of a mathematical structure S , we say that any **object** that conforms to that is an instance of S .

Def 1.48. Group: A group is a **mathematical structure** $\langle G, \circ, e, .^{-1} \rangle$ that consists of:

- a base set G of **objects**,
- an operation $\circ : G \times G \rightarrow G$, such that $\forall a, b \in G. a \circ b \in G$ and $\forall a, b, c \in G. (a \circ b) \circ c = a \circ (b \circ c)$,
- a unit $e \in G$, such that $\forall a \in G. e \circ a = a \circ e = a$, and
- the inverse function $.^{-1} : G \rightarrow G$, such that $\forall a \in G. a \circ a^{-1} = a^{-1} \circ a = e$

An example of a **group** is the set $G = \mathbb{Z}$, with the operation $+ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$; then $e = 0$ and $.^{-1}$ is $\lambda x \in \mathbb{Z}. -x$. We write it as $\langle \mathbb{Z}, +, 0, - \rangle$.

Question: can we do the same for multiplication where $e = 1$? If we think about it, we would be stuck on finding the $.^{-1}$ component of the group. What can we always multiply with any integer and get $e = 1$ back? Take for example 1, we can multiply it with $\frac{1}{1}$ (great!). Take -1 , we can multiply it with $\frac{1}{-1}$ (also great!). But take any other integer, e.g. 2, we must multiply it with $\frac{1}{2}$ to get 1 back, however, $\frac{1}{2} \notin \mathbb{Z}$, it is however $\in \mathbb{Q}$! The multiplicative group would only make sense iff $G = \mathbb{Q} \setminus \{0\}$. We write that as $\langle \mathbb{Q} \setminus \{0\}, *, 1, a \mapsto \frac{1}{a} \rangle$.

But we need some multiplication **structure** that works for \mathbb{Z} !

Def 1.49. Monoid: A monoid is a **mathematical structure** $\langle M, \circ, e \rangle$ that consists of:

- A base set M of **objects**,
- An operation $\circ : M \times M \rightarrow M$, such that for all $a, b \in M$, we have $a \circ b \in M$ (closure),
- Associativity: for all $a, b, c \in M$, $(a \circ b) \circ c = a \circ (b \circ c)$, and
- A unit element $e \in M$, such that for all $a \in M$. $e \circ a = a \circ e = a$.

Hence, the **mathematical structure** $\langle \mathbb{Z}, *, 1 \rangle$ is a **monoid** (a **group** missing an inverse function).

We saw that $\langle \mathbb{Z}, +, 0, - \rangle$ is a **group** under addition. And $\langle \mathbb{Z}, *, 1 \rangle$ is a **monoid** under multiplication. Now, we want to do everyday arithmetics with integers; we want expressions where we have additions and multiplication (e.g. $a * (b + c)$) which evaluates to $(a * b) + (a * c)$.

For that, we need a **mathematical structure** where both addition and multiplication live together.

Def 1.50. Ring: A ring is a **mathematical structure** $\langle R, +, 0, -, *, 1 \rangle$ consisting of:

- A base set R .

- An addition operation $+ : R \times R \rightarrow R$ such that $\langle R, +, 0, - \rangle$ is a **group** (called abelian **group**).
- A multiplication operation $* : R \times R \rightarrow R$ such that $\langle R, *, 1 \rangle$ is a **monoid**.
- Distributivity: for all $a, b, c \in R$, $a * (b + c) = (a * b) + (a * c)$, $(a + b) * c = (a * c) + (b * c)$

Hence, mixing components from the **group** $\langle \mathbb{Z}, +, 0, - \rangle$ and the **monoid** $\langle \mathbb{Z}, *, 1 \rangle$ leaves us with a **ring** $\langle \mathbb{Z}, +, 0, -, *, 1 \rangle$

Def 1.51. Magma: A magma is a **mathematical structure** $\langle M, \circ \rangle$ consisting of:

- A base **set** M , and
- A binary operation $\circ : M \times M \rightarrow M$.

Example: $\langle \mathbb{Z}, - \rangle$ (integers under subtraction). $\langle \mathbb{N}, + \rangle$ (natural numbers under addition)

Def 1.52. Semigroup: A semigroup (**magma** with associativity) is a **mathematical structure** $\langle S, \circ \rangle$ consisting of:

- A base **set** S ,
- A binary operation $\circ : S \times S \rightarrow S$, and
- Associativity: for all $a, b, c \in S$, $(a \circ b) \circ c = a \circ (b \circ c)$

Example: $\langle \mathbb{Z}, + \rangle$ (integers under addition).

1.5 Formal Languages and Grammars

Def 1.53. Alphabet: An alphabet is a **finite set**; we call each element $a \in A$ a **character**, and an n -tuple $s \in A^n$ a **string** (of length n over A). We often write a string $\langle c_1, \dots, c_n \rangle$ as " $c_1 \dots c_n$ ", for instance "abc" for $\langle a, b, c \rangle$

Def 1.54. Empty String: Let A be an **alphabet**, then $A^0 = \{\langle \rangle\}$, where $\langle \rangle$ is the unique 0-tuple. We consider $\langle \rangle$ as the string of length 0 and call it the **empty string** and denote it with ϵ .

Def 1.55. String Length: Given a string s , we denote its length with $|s|$.

Def 1.56. String Concatenation: The concatenation $\text{conc}(s, t)$ of two strings $s = \langle s_1, \dots, s_n \rangle \in A^n$ and $t = \langle t_1, \dots, t_m \rangle \in A^m$ is defined as $s + t$ or simply st

Def 1.57. Kleene Plus/Star: Let A be an **alphabet**, then we define the **sets**:

- $A^+ := \bigcup_{i \in \mathbb{N}^+} A^i$ (*nonempty strings*), and
- $A^* := A^+ \cup \{\epsilon\}$ (*strings*).

Def 1.58. Formal Language: A **set** $L \subseteq A^*$ is called a *formal language* over A . For example, If $A = \{a, b\}$ then:

$$A^* = \bigcup_{n=0}^{\infty} A^n$$

where $A^0 = \epsilon$, $A^1 = \{a, b\}$, $A^2 = \{a, b, ab, ba\}$, and so on ...

$$\begin{aligned} A^* &= \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \dots\} \\ A^+ &= \{a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \dots\} \end{aligned}$$

A **formal language** might be $L = \{a, b, aab, bbb, \dots\}$

Def 1.59. Repeating Chars: We use $c^{[n]}$ for the string that consists of the character c repeated n times. **Examples:**

- Let $A = \{a, b\}$, $a^{[5]} = \text{"aaaaa"}$.
- The **set** $M := \{ba^{[n]} \mid n \in \mathbb{N}\}$ of strings that start with character b followed by an arbitrary numbers of a 's is a **formal language** over $A = \{a, b\}$

Note

A **formal language** might be infinite and even undecidable even if the **alphabet** A is **finite**. For example, let $A = \{a, b\}$ then the **formal language** $L = \{a, ab, bba\}$ is **finite** but the **formal language** $L = \{a^{[n]} \mid n \in \mathbb{N}\}$ is infinite.

Since we cannot list all strings in a **formal language** L because there are infinitely many, we need a **finite description** that can generate all strings in L . We need **Grammars**.

Def 1.60. Phrase Structure Grammar: A phrase structure grammar (*type 0 grammar, unrestricted grammar, grammar*) is a tuple $\langle N, \Sigma, P, S \rangle$ where

- N is a **finite set** of **nonterminal symbols**,
- Σ is a **finite set** of **terminal symbols**. (members of $N \cup \Sigma$ are called symbols).
- P is a **finite set** of **production rules**: pairs $p := h \rightarrow b$ (also written as $h \Rightarrow b$), where $h \in (\Sigma \cup N)^* N (\Sigma \cup N)^*$ and $b \in (\Sigma \cup N)^*$. The string h is called the **head** of p and b the **body**.
- $S \in N$ is a distinguished symbol called the **start symbol** (also **sentence symbol**).

The sets N and Σ are assumed to be **disjoint**. Any word $w \in \Sigma^*$ is called a **terminal word**.

Note

Production rules map strings with at least one **nonterminal** to arbitrary other strings

Notation: If we have n production rule $h \rightarrow b_i$ sharing a head, we often write $h \rightarrow b_1 | \dots | b_n$ instead.

Example: A simple grammar for english sentences:

$$\begin{aligned} S &::= NP\ Vi \\ NP &::= Article\ N \\ Article &::= the\ |\ a\ |\ an \\ N &::= dog\ |\ teacher\ |\ \dots \\ Vi &::= sleeps\ |\ smells\ |\ \dots \end{aligned}$$

Def 1.61. Lexical: A **production rule** whose head is a single **nonterminal** and whose body consists of a single **terminal** is called **lexical** or a **lexical insertion rule**.

Def 1.62. Lexicon of a Grammar: The **subset** of **lexical rules** of a **grammar** G is called the **lexicon** of G

Def 1.63. Vocabulary: The **set** of body symbols are called the **vocabulary** (or the **alphabet**).

Def 1.64. Lexical Categories of a Grammar: The **nonterminals** that appear in the heads of **lexical rules** are called **lexical categories** of the **grammar**.

Def 1.65. Structural Rules: The non **lexicon production rules** are called **structural**.

Def 1.66. Phrasal categories: The **nonterminals** in the heads of **production rules** that expand into other **nonterminals** are called **phrasal** (syntactic) **categories**.

Def 1.67. G-Derivation: Given a **phrase structure grammar** $G := \{N, \Sigma, P, S\}$, we say G **derives** $t \in (\Sigma \cup N)^*$ from $s \in (\Sigma \cup N)^*$ in one step, iff there is a **production rule** $p \in P$ with $p = h \rightarrow b$ and there are $u, v \in (\Sigma \cup N)^*$, such that $s = uhv$ and $t = ubv$. We write $s \xrightarrow{p} t$ (or $s \xrightarrow{G} t$ if p is clear from the context) and use $\xrightarrow{*}$ for the **reflexive transitive closure** of \xrightarrow{G} . We call $s \xrightarrow{*} t$ a G -derivation of t from s .

Def 1.68. Sentential Form: Given a **phrase structure grammar** $G := \{N, \Sigma, P, S\}$, we say that $s \in (\Sigma \cup N)^*$ is a **sentential form** of G , iff $S \xrightarrow{*} s$

Def 1.69. Sentence: A **sentential form** that does not contain **nonterminals** is called a **sentence** of G , we also say that G **accepts** s . We say that G **rejects** s , iff it is not a sentence of G .

Def 1.70. Language Generation: The language $L(G)$ of G is the **set** of its **sentences**. We say that $L(G)$ is **generated** by G .

Def 1.71. Equivalent Grammars: We call two **grammars** **equivalent**, iff they have the same **languages**.

Def 1.72. Universal Grammar: A **grammar** G is said to be **universal** if $L(G) = \Sigma^*$

Def 1.73. Syntactic Analysis: Syntactic analysis (parsing / syntax analysis) is the process of analyzing a string of symbols, either in a **formal** or a natural language by means of a **grammar**.

Note

The shape of the grammar determines the size of its language

Def 1.74. Context-Sensitive: We call a grammar context-sensitive (or type 1), if the bodies of production rules have no less symbols than the heads.

Def 1.75. Context-Free: We call a grammar context-free (or type 2), iff the heads have exactly one symbol.

Def 1.76. Regular: We call a grammar regular (or type 3, or REG), if additionally the bodies are empty or consist of a nonterminal, optionally followed by a terminal symbol.

Note

By extension, a formal language L is called *context-sensitive / context-free / regular*, iff it is the language of a respective grammar. Context-free grammars are sometimes CFGs and context-free languages CFLs.

1.6 Graphs and Trees

Def 1.77. Undirected Graph: An undirected graph is a pair $\langle V, E \rangle$ such that

- V is a set of vertices (nodes), and
- $E \subseteq \{ \{v, \bar{v}\} \mid v, \bar{v} \in V \wedge v \neq \bar{v} \}$ is the set of its undirected edges.

Def 1.78. Directed Graph: A directed graph (digraph) is a pair $\langle V, E \rangle$ such that

- V is a set of vertices (nodes), and
- $E \subseteq V \times V$ is the set of its directed edges.

Def 1.79. Indegree: Given a directed graph $\langle V, E \rangle$, the indegree $\text{indeg}(v)$ of a vertex $v \in V$ is defined as $\text{indeg}(v) = \Gamma(\{ w \mid (w, v) \in E \})$.

Def 1.80. Outdegree: Given a directed graph $\langle V, E \rangle$, the outdegree $\text{outdeg}(v)$ (branching factor) of a vertex $v \in V$ is defined as $\text{outdeg}(v) = \Gamma(\{ w \mid (v, w) \in E \})$.

Note

directed graphs are nothing else than relations.

Def 1.81. Initial vs Terminal Node: Let $G = \langle V, E \rangle$ be a directed graph, then we call a node $v \in V$

- **initial** (source of G), iff there is no $w \in V$ such that $(w, v) \in E$ (no predecessor)
- **terminal** (sink of G), iff there is no $w \in V$ such that $(v, w) \in E$ (no successor)

Def 1.82. Graph Isomorphism: Iff we can find a bijection $\psi : V \rightarrow \bar{V}$ between two graphs $G = \langle V, E \rangle$ and $\bar{G} = \langle \bar{V}, \bar{E} \rangle$ then we call them isomorphic. The bijection $\psi : V \rightarrow \bar{V}$ is defined as $(a, b) \in E \Leftrightarrow (\psi(a), \psi(b)) \in \bar{E}$ for directed graph. And for undirected graph it is defined as $\{a, b\} \in E \Leftrightarrow \{\psi(a), \psi(b)\} \in \bar{E}$

Def 1.83. Equivalent Graphs: Two graphs G and \bar{G} are equivalent iff there is an isomorphism ψ between G and \bar{G} .

Def 1.84. Labeled Graph: A labeled graph G is a quadruple $\langle V, E, L, l \rangle$ where $\langle V, E \rangle$ is a graph and $l : V \cup E \rightarrow L$ is a partial function into a set L of labels.

Def 1.85. Paths in Graphs: Given a graph $G := \langle V, E \rangle$ we call a $n + 1$ -tuple $p = \langle v_0, \dots, v_n \rangle \in V^{n+1}$ a path in G iff $(v_{i-1}, v_i) \in E$ for all $1 \leq i \leq n$ and $n > 0$.

- We say that v_i are nodes on p and that v_0 and v_n are linked by p .
- v_0 and v_n are called the start and end of p (write $\text{start}(p)$ and $\text{end}(p)$), the other v_i are called inner nodes of p .
- n is called the length of p (write $\text{len}(p)$).
- We denote the set of paths in G with $\Pi(G)$.

Def 1.86. Cyclic Graphs: Given a [directed graph](#) $G = \langle V, E \rangle$, a [path](#) p is called **cyclic** iff $\text{start}(p) = \text{end}(p)$. A cycle $\langle v_0, \dots, v_n \rangle$ is called **simple**, iff $v_i \neq v_j$ for $1 \leq i, j \leq n$ with $i \neq j$ (all inner nodes are distinct).

Def 1.87. DAG: A [directed graph](#) with no [cycles](#) is called **directed acyclic graph (DAG)**.

Def 1.88. Node Depth: Let $G = \langle V, E \rangle$ be a [directed graph](#), then the depth $\text{dp}(v)$ of a vertex $v \in V$ is defined to be 0, iff v is a source of G and the supremum $\sup(\{\text{len}(p) \mid \text{indeg}(\text{start}(p)) = 0 \wedge \text{end}(p) = v\})$ otherwise, i.e. the length of the longest [path](#) from a source of G to v (can be infinite).

Def 1.89. Graph Depth: Given a [directed graph](#) $G = \langle V, E \rangle$, the **depth** ($\text{dp}(G)$) of G is defined as the supremum $\sup(\{\text{len}(p) \mid p \in \Pi(G)\})$, i.e. the maximal path length in G .

Def 1.90. Tree: A tree is a [DAG](#) $G = \langle V, E \rangle$ such that

- There is exactly one [initial node](#) $v_r \in V$ (called the **root**)
- All nodes but the root have [indegree](#) of 1.

We call v the **parent** of w , iff $(v, w) \in E$ (w is a child of v). We call a node v a **leaf** of G , iff it is [terminal](#), i.e. if it does not have children.

Note

For any node $v \in V$ except the root v_r , there is exactly one [path](#) $p \in \Pi(G)$ with $\text{start}(p) = v_r$ and $\text{end}(p) = v$.

2 Introduction

Def 2.1. Artificial intelligence (AI): The capability of computational systems to perform tasks typically associated with human intelligence, such as learning, reasoning, problem-solving, perception, and decision-making

Def 2.2. Symbolic AI: A subfield of [Artificial Intelligence \(AI\)](#) based on the assumption that many aspects of intelligence can be achieved by the manipulation of symbols, combining them into meaningful structures (expressions) and manipulating them (using processes) to produce new expressions.

Def 2.3. Statistical AI: Remedies the two shortcomings of [symbolic AI](#) approaches: that all concepts represented by symbols are crisply defined, and that all aspects of the world are knowable/representable in principle. Statistical AI adopts sophisticated mathematical models of uncertainty and uses them to create more accurate world models and reason about them.

Def 2.4. Subsymbolic AI (a.k.a connectionism/neural AI): A subfield of [AI](#) that posits that intelligence is inherently tied to brains, where information is represented by a simple sequence pulses that are processed in parallel via simple calculations realized by neurons, and thus concentrates on neural computing.

Def 2.5. Embodied AI: Posits that intelligence cannot be achieved by [reasoning](#) about the state of the world ([symbolically](#), [statistically](#), or [sub-symbolically](#)), but must be embodied i.e. situated in the world, equipped with a "body" that can interact with it via sensors and actuators. Here, the main method for realizing intelligent behavior is by learning from the world.

Def 2.6. Reasoning: The process of producing valid arguments and predictive world models. There are three forms of reasoning:

- **deductive reasoning** to produce new knowledge from existing knowledge. *Example:* All humans are mortal. Socrates is a human. Therefore, Socrates is mortal.
- **inductive reasoning** to produce knowledge from perception. *Example:* The sun has risen every morning in recorded history. Therefore, the sun will rise tomorrow.
- **abductive reasoning** to produce explanations for observations and given knowledge. *Example:* The grass is wet this morning. If it rained last night, that would explain it. Therefore, it probably rained.

Def 2.7. Inference: The act or process of reaching a **conclusion** about something from known facts or evidence (jointly called **premises**)

Def 2.8. Formal Logic: The science of [deductively](#) valid inferences, meaning arguments whose [conclusions](#) necessarily follow from their [premises](#) by virtue of their form (their structure), regardless of the topic.

Def 2.9. Agent: An agent is a [structure](#) $A := \langle \mathcal{P}, \mathcal{A}, f \rangle$ where:

- \mathcal{P} is a [set](#) of percepts
- \mathcal{A} is a [set](#) of actions
- f is a [function](#) $f : \mathcal{P}^* \rightarrow \mathcal{A}$ that maps from percepts to actions.

In other words, an agent is anything that perceives its environment via sensors and acts on it with actuators.

Def 2.10. Performance Measure: A [function](#) that evaluates a sequence of environments.

Def 2.11. Rational: An [agent](#) is called **rational**, if it chooses whichever action maximizes the expected value of the [performance measure](#) given the percept sequence to date.

Def 2.12. PEAS: To design [rational agents](#), we must specify the PEAS components: [Performance Measure](#), Environment, Actuators, and Sensors.

Def 2.13. Environment Types: For an [agent](#) a we classify the environment e of a by its **type**, which is one of the following. We call e

1. *fully observable*, iff the a 's sensors give it access to the complete state of the environment at any point in time; otherwise, we call it *partially observable*.
2. *deterministic*, iff the next state of the environment is completely determined by the current state and a 's actions; otherwise, *stochastic*.
3. *episodic*, iff a 's experience is divided into atomic episodes, where it perceives and then performs a single action, and the next episode does not depend on previous ones. Otherwise, *sequential*.
4. *dynamic*, iff the environment can change without an action performed by a ; otherwise, *static*. If the environment does not change but a 's performance measure does, we call e *semidynamic*.

5. *discrete*, iff the sets of e 's state and a 's actions are countable; otherwise, *continuous*.

6. *single-agent*, iff only a acts on e ; otherwise *multi-agent*.

Def 2.14. Reflex: An agent $\langle \mathcal{P}, \mathcal{A}, f \rangle$ is called a **reflex agent**, iff it only takes the last percept into account when choosing an action, i.e. $f(p_1, \dots, p_k) = f(p_k) \forall p_1 \dots p_k \in \mathcal{P}$

Def 2.15. Model-Based Agent: A model-based agent $\langle \mathcal{P}, \mathcal{A}, \mathcal{S}, \mathcal{T}, s_0, S, a \rangle$ is an agent $\langle \mathcal{P}, \mathcal{A}, f \rangle$ whose actions depend on:

- a **world model**: a set \mathcal{S} of possible states, and a start state $s_0 \in \mathcal{S}$.
- a **transition model** \mathcal{T} that predicts a new state $\mathcal{T}(s, a)$ from a state s and an action a .
- a **sensor model** S that given a state s and a percept p determines a new state $S(s, p)$.
- an **action function** $a : \mathcal{S} \rightarrow \mathcal{A}$ that given a state $s \in \mathcal{S}$ selects the next action $a \in \mathcal{A}$.

Note

If the agent is in state s then it took action a and now perceives p , then the agent state will become $s' = S(p, \mathcal{T}(s, a))$ and accordingly take action $a' = a(s')$.

Def 2.16. Goal Based Agent: A goal based agent $\langle \mathcal{P}, \mathcal{A}, \mathcal{S}, \mathcal{T}, s_0, S, a, \mathcal{G} \rangle$ is a model based agent with an explicit set of goals. It consists of:

- a set of internal states \mathcal{S} and an initial state $s_0 \in \mathcal{S}$,
- a transition model $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$,
- a state update function $S : \mathcal{P} \times \mathcal{S} \rightarrow \mathcal{S}$,
- a set of goals \mathcal{G} ,
- a goal conditioned action function $a : \mathcal{S} \times \mathcal{G} \rightarrow \mathcal{A}$ selecting an action given a state and the goals.

Def 2.17. Utility-Based Agent: A utility-based agent uses a world model along with a utility function that models its preferences among the states of that world. It chooses the action that maximizes the expected utility.

Def 2.18. Learning Agent: A learning agent is an agent that can improve its own behavior through experience. It is composed of four main components:

- the performance element, which selects actions based on the agent's current percepts and represents its existing knowledge or behavior,
- the learning element, which improves the performance element over time by analyzing feedback and identifying how to make better decisions,
- the critic, which evaluates the agent's performance according to a given performance standard and provides feedback to the learning element,
- the problem generator, which suggests new and informative actions or experiences that help the agent explore and learn more effectively.

Def 2.19. State Representation: We call a state representation **atomic**, iff it has no internal structure (black box). However, iff each state is characterized by attributes and their values then the representation is **factored**. A **structured** state representation is when include representations of objects, their properties and relationships.

3 General Problem Solving

Def 3.1. Search Problem: A search problem $\Pi := \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{I}, \mathcal{G} \rangle$ consists of a set \mathcal{S} of states, a set \mathcal{A} of actions, and a transition model $\mathcal{T} : \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{P}(\mathcal{S})$ that assigns to any action $a \in \mathcal{A}$ and state $s \in \mathcal{S}$ a set of successor states. Certain states in \mathcal{S} are designated as goal (terminal) states ($\mathcal{G} \subseteq \mathcal{S}$ with $\mathcal{G} \neq \emptyset$) and initial states $\mathcal{I} \subseteq \mathcal{S}$.

Def 3.2. Action Application: We say that an action $a \in \mathcal{A}$ is applicable in state $s \in \mathcal{S}$, iff $\mathcal{T}(a, s) \neq \emptyset$ and that any $s' \in \mathcal{T}(a, s)$ is a result of applying action a to state s . We call $\mathcal{T}_a : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{S})$ with $\mathcal{T}_a(s) := \mathcal{T}(a, s)$ the result relation for a and $\mathcal{T}_{\mathcal{A}} := \bigcup_{a \in \mathcal{A}} \mathcal{T}_a$ the result relation of Π .

Def 3.3. State Space: The graph $\langle \mathcal{S}, \mathcal{T}_{\mathcal{A}} \rangle$ is called the state space induced by Π .

Def 3.4. Solution: A solution for Π consists of a sequence a_1, \dots, a_n of actions such that for all $1 < i \leq n$:

- a_i is applicable to state s_{i-1} where $s_0 \in \mathcal{I}$, and
- $s_i \in \mathcal{T}_{a_i}(s_{i-1})$, and $s_n \in \mathcal{G}$

Def 3.5. Cost Function: Often we add a cost function $c : \mathcal{A} \rightarrow \mathbb{R}_0^+$ that associates a step cost $c(a)$ to an action $a \in \mathcal{A}$. The cost of a solution is the sum of the step costs of its actions.

Note

- For deterministic environments, we have $|\mathcal{T}(a, s)| \leq 1$
- For fully observable ones, we have $\mathcal{I} = \{s_0\}$

Def 3.6. Successor Function/State: In a search problem, \mathcal{T}_a induces a partial function $S_a : \mathcal{S} \rightharpoonup \mathcal{S}$ whose natural domain is the set of states where a is applicable: $S_a(s) := s'$ if $\mathcal{T}_a = \{s'\}$ and undefined at s otherwise. We call S_a the successor function for a and $S_a(s)$ the successor state of s .

Note

- A search problem is called a single-state problem iff it is fully observable, deterministic, static, and discrete.
- A search problem is called a multi-state problem iff it is partially observable.
- A search problem is called a contingency problem iff the environment is non deterministic and the state space is unknown.

Def 3.7. Tree Search: Given a search problem $\Pi := \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{I}, \mathcal{G} \rangle$, the tree search algorithm consists of the simulated exploration of the state space $\langle \mathcal{S}, \mathcal{T}_{\mathcal{A}} \rangle$ in a search tree formed by successively expanding already explored states.

Def 3.8. Path Cost: We define the path cost of a node n in a search tree T to be the sum of the step costs on the path from n to the root of T .

The general structure of Tree Search algorithms is as follows:

```
TREE_SEARCH(start, INSERT):
    fringe ← empty list
    append start to fringe
    while fringe is not empty:
        node ← remove first element of fringe
        if is_goal(node):
            return node
        children ← expand(node)
        INSERT(fringe, children)
    return failure
```

Def 3.9. Strategy: A strategy is a function that picks a node from the fringe of a search tree

Note

Every algorithm has its own strategy, hence the implementation of the INSERT differs based on that strategy.

Def 3.10. Properties of Strategies:

- completeness: does it always find a solution if one exists?
- optimality: does it always find the optimal solution (least cost)?

- time complexity: number of nodes the algorithm explores (expands)
- space complexity: maximum number of nodes in memory

Note

Time and space complexity measured in terms of:

- b : maximum branching factor of the search tree
- d : minimal graph depth of a solution in the search tree
- m : maximum graph depth of the search tree

Def 3.11. Breadth-First Search (BFS): The BFS strategy treats the fringe as a FIFO queue, i.e. successors go in at the end (back) of the fringe.

BFS Properties Analysis

- Time Complexity:
assume we have a binary search tree ($b = 2$) and we found solution in depth = 3. In the 0th level we explored $2^0 = 1$ nodes. In the 1st level, $2^1 = 2$ nodes. In the 2nd level, $2^2 = 4$ nodes. And finally $2^3 = 8$ nodes in the third level. Number of nodes the algorithm explored will be $2^0 + 2^1 + 2^2 + 2^3$.
 - Generally, for arbitrary b, d , we have $1 + b + b^2 + \dots + b^d$, which means, worst time complexity is $\mathcal{O}(b^d)$ (exponential in d).
- The space complexity is also $\mathcal{O}(b^d)$.
- **BFS** is complete if: (i) b is finite, and (ii) the state space is finite or has a solution.
- **BFS** is optimal if we have a uniform constant cost for all actions.

Def 3.12. Uniform-Cost Search (UCS): UCS is the strategy where the fringe is ordered by increasing path cost. (*expands least cost first*)

Note

UCS it is equivalent to BFS if all costs are equal

UCS Properties Analysis

- Time and space complexity $\approx \mathcal{O}(b^d)$
- **UCS** is complete if:
 - b is finite
 - all actions costs are $\geq \epsilon > 0$
 - state space is finite or has a solution.

Def 3.13. Depth-First Search (DFS): DFS is the strategy where the fringe is organized as a LIFO stack, i.e. successors go in at the front of the fringe.

Def 3.14. Backtracking: Every node that is pushed to the stack is called a **backtrack point**. Popping a non-goal node from the stack and continuing the search with the new top element is called **backtracking**. For this reason, the **DFS algorithm** is also referred to as a **backtracking search**.

DFS Properties Analysis

- Time complexity is $\mathcal{O}(b^m)$ (exponential in the maximum depth)
- Space complexity is linear. While **BFS** keeps track of all nodes at the current level in memory, **DFS** needs to only keep the current path (from root to m), and also needs to keep the remaining unexplored sibling root node for each node along that path (to backtrack). This means, for each level, we store the siblings, i.e. b nodes. We explore until we hit the max depth m and in each level we store b . Hence we have a linear space $\mathcal{O}(b \cdot m)$
- **DFS** is complete if the search tree is finite and acyclic.
- **DFS** is not optimal.

Def 3.15. Depth-Limited Search (DLS): DLS is a [DFS](#) with a depth limit l . We treat all nodes at depth l as if they have no children.

DLS Properties Analysis

- Time complexity $\sim \mathcal{O}(b^l)$
- Space complexity $\sim \mathcal{O}(b \cdot l)$
- Not complete (solution maybe beyond chosen l)
- Not optimal

Def 3.16. Iterative Deepening Search (IDS): IDS is a [DLS](#) with an ever-increasing depth limit. We call the difference between successive depth limits the step size.

Note

- [IDS](#) solves the problem of choosing a good l in [DLS](#).
- [IDS](#) tries all values until either a [solution](#) is found (depth is returned to [DLS](#), we call that the cutoff value) or we return failure.
- [IDS](#) combines benefits of [DFS](#) and [BFS](#).

IDS Properties Analysis

It is similar to [DFS](#) in memory requirements (assuming [finite](#) and [acyclic search tree](#)).

- $\mathcal{O}(b \cdot d)$ when there is a [solution](#)
- $\mathcal{O}(b \cdot m)$ when there is no [solution](#)

It is similar to [BFS](#) in terms of optimality, completeness, and time complexity. Optimal for problems where all actions have the same cost. Complete if b is [finite](#) and the [state space](#) is finite or has a solution.

With regard to time complexity, consider [IDS](#) finds a solution at depth d , then:

- nodes at the very bottom (d) are visited only once in the final iteration.
- nodes at $d - 1$ are visited twice, once in the search with depth $d - 1$ and once in the search with depth d .
- nodes at $d - 2$ are visited three times
- and so on, until root node ($d = 0$) which is visited d times.

Generally, we have $(d)b^1 + (d-1)b^2 + (d-2)b^3 + \dots + b^d$. However, the majority of nodes are at the max depth where there are b^d nodes. Those are only visited once and the extra cost of visiting shallower nodes does not significantly increase the overall count. Therefore, time complexity is bounded by b^d ($\mathcal{O}(b^d)$) because b^d dominates the total nodes count in a large [search space](#).

Def 3.17. Graph Search: A graph search algorithm is a variant of a [tree search algorithm](#) that prunes nodes whose state has already been considered (duplicate pruning), essentially using a [DAG](#) data structure.

The general structure of [Graph Search](#) algorithms is as follows:

```
GRAPH_SEARCH(start, INSERT):
    fringe ← empty list
    visited ← empty set
    append start to fringe
    add state(start) to visited
    while fringe is not empty:
        node ← remove first element of fringe
        if is_goal(node):
            return node
        children ← expand(node)
        for each child in children:
            if state(child) not in visited:
                add state(child) to visited
                INSERT(fringe, child)
    return failure
```

Def 3.18. Search Alrogrithm: We speak of a **search algorithm** when we do not want to distinguish whether it is a **tree** or a **graph search algorithm** - *difference considered as an implementation detail.*

Def 3.19. Informed: A **search algorithm** is called **informed**, iff it uses some form of external information to guide the search.

Def 3.20. Evaluation Function: An evaluation function assigns a desirability value to each node of the **search tree**. It is not part of the **search problem**, but must be added externally.

Def 3.21. Best-First Search: In best-first search, the fringe is a queue sorted in decreasing order of desirability.

Def 3.22. Heuristic: A heuristic is an **evaluation function** h on states that estimates the **cost** from n to the nearest goal state. We speak of **heuristic search** if the **search algorithm** uses a heuristic in some way.

Def 3.23. Greedy Search: A **best-first search strategy** where the fringe is organized as a queue sorted based on h value.

Greedy Search Properties

- Time and space complexity are both exponential in m (max depth) $\rightsquigarrow \mathcal{O}(b^m)$
- It is not optimal. But it is complete if the **state space** is finite.

Def 3.24. Heuristic Function: A heuristic funciton for **search problem** Π is a **function** $h : \mathcal{S} \rightarrow \mathbb{R}_0^+ \cup \{\infty\}$ so that $h(s) = 0 \quad \forall s \in \mathcal{G}$.

Def 3.25. Goal Distance Function: The **function** $h^* : \mathcal{S} \rightarrow \mathbb{R}_0^+ \cup \{\infty\}$, where $h^*(s)$ is the **cost** of a cheapest **path** from s to a goal state, or ∞ if no such path exists. (*the perfect heuristic which we do not know and cannot compute*)

Def 3.26. Admissible: For a **search problem** Π with states \mathcal{S} and actions \mathcal{A} . We say that a **heuristic** h for Π is **admissible** if

$$h(s) \leq h^*(s) \quad \forall s \in \mathcal{S}$$

Note

Admissible heuristics never overestimates; our guess should be always optimistic or exact, never too high

Def 3.27. Consistent: For a **search problem** Π with states \mathcal{S} and actions \mathcal{A} . We say that a **heuristic** h for Π is **consistent** if

$$h(s) - h(s') \leq c(a) \quad \forall s \in \mathcal{S}, a \in \mathcal{A}, s' \in \mathcal{T}(s, a)$$

Note

A **consistent heuristic** never drops faster than the real cost you pay when moving through the state space. Formally, for every transition $s \xrightarrow{a} s'$,

$$h(s) \leq c(a) + h(s')$$

This means the heuristic respects a triangle inequality with respect to the actual **step costs**. Consistency ensures that the estimated **total cost** along a **path** never decreases.

Def 3.28. A* Evaluation Function: Given a **path cost** function g and a **heuristic** h , the **A* evaluation function** is

$$f(n) = g(n) + h(n).$$

It estimates the total cost of a cheapest path from the **start state** to a goal via n .

Def 3.29. A* Search: A* search is the **best-first search** strategy that uses the **A* evaluation function** $f = g + h$ to order the fringe.

Theorem 3.1. A* Search with **admissible heuristic** is optimal

Def 3.30. Dominance: Let h_1 and h_2 be two **admissible heuristic**, we say that h_2 **dominates** h_1 if $h_2(s) \geq h_1(s) \quad \forall s \in \mathcal{S}$.

Theorem 3.2. If h_2 **dominates** h_1 , then h_2 is better for **search** than h_1

Proof. If h_2 dominates h_1 , then h_2 is closer to h^* than h_1 □

Def 3.31. Adversarial Search: An adversarial search problem is a [search problem](#) $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{I}, \mathcal{G} \rangle$, where:

- $\mathcal{S} = \mathcal{S}^{\text{Max}} \uplus \mathcal{S}^{\text{Min}} \uplus \mathcal{G}$
- $\mathcal{A} = \mathcal{A}^{\text{Max}} \uplus \mathcal{A}^{\text{Min}}$
- For $a \in \mathcal{A}^{\text{Max}}$, if $s \xrightarrow{a} s'$, then $s \in \mathcal{S}^{\text{Max}}$ and $s' \in (\mathcal{S}^{\text{Min}} \cup \mathcal{G})$
- For $a \in \mathcal{A}^{\text{Min}}$, if $s \xrightarrow{a} s'$, then $s \in \mathcal{S}^{\text{Min}}$ and $s' \in (\mathcal{S}^{\text{Max}} \cup \mathcal{G})$

together with a **game utility function** $u : \mathcal{G} \rightarrow \mathbb{R}$

Def 3.32. Strategy: Let Θ be an [adversarial search problem](#), and let $X \in \{\text{Max}, \text{Min}\}$. A **strategy** for X is a [function](#) $\sigma^X : \mathcal{S}^X \rightarrow \mathcal{A}^X$ so that $a \in \mathcal{A}^X$ is [applicable](#) to $s \in \mathcal{S}^X$ whenever $\sigma^X(s) = a$.

Def 3.33. Optimal Strategy: A [strategy](#) is called **optimal** if it yields the best possible [utility](#) for X assuming perfect opponent play.

Note

Assumptions for Adversarial Games:

- Two players acting in strictly alternating turns
- Fully observable and deterministic game states
- Finite state space so the game tree is finite
- Terminal states have a real-valued utility
- The game is zero-sum: Max maximizes the utility, Min minimizes the same utility

Def 3.34. Minimax: Let Θ be an [adversarial search problem](#) with [utility function](#) $u : \mathcal{G} \rightarrow \mathbb{R}$. The **minimax value** is the [function](#) $\hat{u} : \mathcal{S} \rightarrow \mathbb{R}$ defined by:

$$\hat{u}(s) = \begin{cases} u(s) & \text{if } s \in \mathcal{G}, \\ \max_{s' \in \mathcal{T}(a,s)} \hat{u}(s') & \text{if } s \in \mathcal{S}^{\text{Max}}, \\ \min_{s' \in \mathcal{T}(a,s)} \hat{u}(s') & \text{if } s \in \mathcal{S}^{\text{Min}}, \end{cases}$$

where a ranges over all actions [applicable](#) to s . The **minimax decision** at a root state r is any action whose [successor state](#) achieves $\hat{u}(r)$.

Def 3.35. Minimax Algorithm: The minimax algorithm is given by the following recursive function whose arguments are a node (state) and whether it is Max's turn or Min's.

```
function MINIMAX(node, maximizing):
    if TERMINAL(node):
        return UTILITY(node)

    if maximizing:
        best = -infinity
        for each child in SUCCESSORS(node):
            val = MINIMAX(child, false)
            if val > best:
                best = val
        return best

    else:
        best = +infinity
        for each child in SUCCESSORS(node):
            val = MINIMAX(child, true)
            if val < best:
                best = val
        return best
```

Def 3.36. Evaluation Function: In practice, full-depth [minimax](#) is infeasible because the [search tree](#) is too large. We therefore impose a fixed *depth limit / horizon* d and stop the search at all states $\{s \in \mathcal{S} \mid \text{dp}(s) = d\}$, called *cut-off states*.

An evaluation function is a [function](#) $f : \mathcal{S} \rightarrow \mathbb{R}$ that assigns a numerical estimate of the true [minimax value](#) $\hat{u}(s)$ to any nonterminal cut-off state s . If a cut-off state is terminal, its utility $u(s)$ is used instead of $f(s)$.

Def 3.37. Alpha Value: For each node n in a [minimax search tree](#), the alpha value $\alpha(n)$ is the highest Max-node utility that search has encountered on its [path](#) from the [root](#) to n .

Def 3.38. Beta Value: For each node n in a [minimax search tree](#), the beta value $\beta(n)$ is the highest Min-node utility that search has encountered on its [path](#) from the [root](#) to n .

Def 3.39. Alpha-Beta Pruning: Alpha-beta pruning is a variant of [minimax](#) that avoids exploring branches that cannot change the [minimax value](#).

- At a Max-node n , if a successor s is found with utility $\hat{u}(s) \geq \beta(n)$, then no remaining successors of n can affect its value and they are pruned.
- At a Min-node n , if a successor s is found with utility $\hat{u}(s) \leq \alpha(n)$, then no remaining successors of n can affect its value and they are pruned.

The algorithm returns the same [minimax value](#) as full search while expanding fewer nodes.

```
function ALPHABETA(node, alpha, beta, maximizing):
    if TERMINAL(node):
        return UTILITY(node)

    if maximizing:
        best = -infinity
        for each child in SUCCESSORS(node):
            val = ALPHABETA(child, alpha, beta, false)
            if val > best:
                best = val
            if best > alpha:
                alpha = best
            if beta <= alpha:
                break
        return best

    else:
        best = +infinity
        for each child in SUCCESSORS(node):
            val = ALPHABETA(child, alpha, beta, true)
            if val < best:
                best = val
            if best < beta:
                beta = best
            if beta <= alpha:
                break
        return best
```