Fundamentals of Artificial Intelligence and Knowledge Representation

Last update: 12 October 2023

Academic Year 2023 – 2024
Alma Mater Studiorum · University of Bologna

Contents

Not required for the exam

1 Introduction

1.1 AI systems classification

1.1.1 Intelligence classification

Intelligence is defined as the ability to perceive or infer information and to retain the knowledge for future use.

Weak AI aims to build a system that acts as an intelligent system. Weak AI

Strong AI aims to build a system that is actually intelligent. Strong AI

1.1.2 Capability classification

General AI systems able to solve any generalized task. General AI

Narrow AI systems able to solve a particular task. Narrow AI

1.1.3 AI approaches

Symbolic AI (top-down) Symbolic representation of knowledge, understandable by humans. Symbolic AI

Connectionist approach (bottom up) Neural networks. Knowledge is encoded and not understandable by humans. Connectionist approach

1.2 Symbolic AI

Deductive reasoning Conclude something given some premises (general to specific). It is unable to produce new knowledge. Deductive reasoning

Example. "All men are mortal" and "Socrates is a man" → "Socrates is mortal"

Inductive reasoning A conclusion is derived from an observation (specific to general). Produces new knowledge, but correctness is not guaranteed. Inductive reasoning

Example. "Several birds fly" → "All birds fly"

Abduction reasoning An explanation of the conclusion is found from known premises. Differently from inductive reasoning, it does not search for a general rule. Produces new knowledge, but correctness is not guaranteed. Abduction reasoning

Example. "Socrates is dead" (conclusion) and "All men are mortal" (knowledge) → "Socrates is a man"

Reasoning by analogy Principle of similarity (e.g. k-nearest-neighbor algorithm). Reasoning by analogy

Example. "Socrates loves philosophy" and Socrates resembles John → "John loves philosophy"

Constraint reasoning and optimization Constraints, probability, statistics. Constraint reasoning

Not required for the exam

1.3 Machine learning

1.3.1 Training approach

Supervised learning

Supervised learning Trained on labeled data (ground truth is known). Suitable for classification and regression tasks.

Unsupervised learning

Unsupervised learning Trained on unlabeled data (the system makes its own discoveries). Suitable for clustering and data mining.

Semi-supervised learning

Semi-supervised learning The system is first trained to synthesize data in an unsupervised manner, followed by a supervised phase.

Reinforcement learning

Reinforcement learning An agent learns by simulating actions in an environment with rewards and punishments depending on its choices.

1.3.2 Tasks

Classification

Classification Supervised task that, given the input variables X and the output (discrete) categories Y, aims to approximate a mapping function $f : X \to Y$.

Regression

Regression Supervised task that, given the input variables X and the output (continuous) variables Y, aims to approximate a mapping function $f : X \to Y$.

Clustering

Clustering Unsupervised task that aims to organize objects into groups.

1.3.3 Neural networks

Perceptron

A neuron (**perceptron**) computes a weighted sum of its inputs and passes the result to an activation function to produce the output.

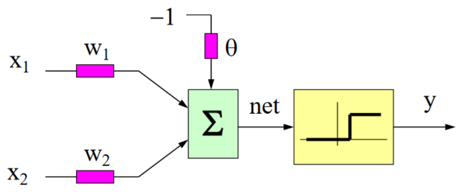


Figure 1.1: Representation of an artificial neuron

Feed-forward neural network

A **feed-forward neural network** is composed of multiple layers of neurons, each connected to the next one. The first layer is the input layer, while the last is the output layer. Intermediate layers are hidden layers.
The expressivity of a neural networks increases when more neurons are used:

Single perceptron Able to compute a linear separation.

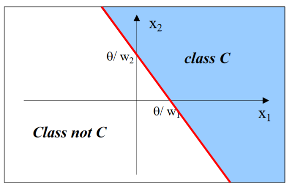


Figure 1.2: Separation performed by one perceptron

Not required for the exam

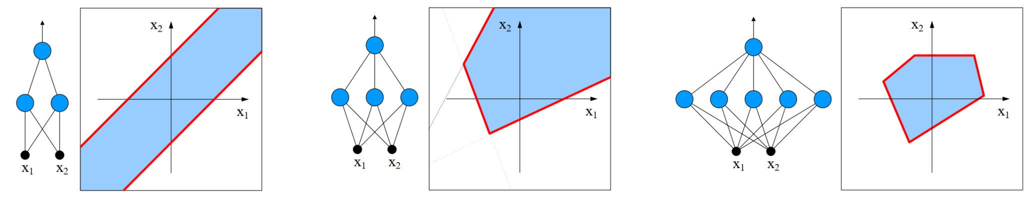


Figure 1.3: Separation performed by a three-layer network

Three-layer network Able to separate a convex region ($n_{\text{edges}} \leq n_{\text{hidden neurons}}$)

Four-layer network Able to separate regions of arbitrary shape.

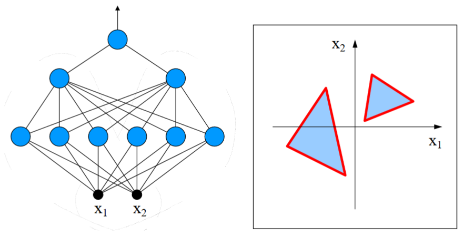


Figure 1.4: Separation performed by a four-layer network

Theorem 1.3.1 (Universal approximation theorem)**.** A feed-forward network with one hidden layer and a finite number of neurons is able to approximate any continuous function with desired accuracy.

Universal approximation theorem

Deep learning Neural network with a large number of layers and neurons. The learning process is hierarchical: the network exploits simple features in the first layers and synthesis more complex concepts while advancing through the layers.

Deep learning

1.4 Automated planning

Given an initial state, a set of actions and a goal, **automated planning** aims to find a partially or totally ordered sequence of actions to achieve a goal.

Automated planning

An **automated planner** is an agent that operates in a given domain described by:

- Representation of the initial state
- Representation of a goal
- Formal description of the possible actions (preconditions and effects)

1.5 Swarm intelligence

Decentralized and self-organized systems that result in emergent behaviors.

Swarm intelligence

1.6 Decision support systems

Knowledge based system Use knowledge (and data) to support human decisions. Bottlenecked by knowledge acquisition.

Knowledge based system

Not required for the exam

Different levels of decision support exist:

Descriptive analytics Data are used to describe the system (e.g. dashboards, reports, ...). Human intervention is required.

Descriptive analytics

Diagnostic analytics Data are used to understand causes (e.g. fault diagnosis) Decisions are made by humans.

Diagnostic analytics

Predictive analytics Data are used to predict future evolutions of the system. Uses machine learning models or simulators (digital twins)

Predictive analytics

Prescriptive analytics Make decisions by finding the preferred scenario. Uses optimization systems, combinatorial solvers or logical solvers.

Prescriptive analytics

2 Search problems

2.1 Search strategies

Solution space Set of all the possible sequences of actions an agent may apply. Some of these lead to a solution. Solution space

Search algorithm Takes a problem as input and returns a sequence of actions that solves the problem (if exists). Search algorithm

2.1.1 Search tree

Expansion Starting from a state, apply a successor function and generate a new state. Expansion

Search strategy Choose which state to expand. Usually is implemented using a fringe that decides which is the next node to expand. Search strategy

Search tree Tree structure to represent the expansion of all states starting from a root (i.e. the representation of the solution space). Search tree

Nodes are states and branches are actions. A leaf can be a state to expand, a solution or a dead-end. Algorithm 1 describes a generic tree search algorithm.

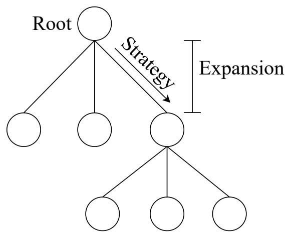


Figure 2.1: Search tree

Each node contains:

- The state
- The parent node
- The action that led to this node
- The depth of the node
- The cost of the path from the root to this node

2.1.2 Strategies

Non-informed strategy Domain knowledge not available. Usually does an exhaustive search. Non-informed strategy

Informed strategy Use domain knowledge by using heuristics. Informed strategy

Algorithm 1 Tree search

```
def treeSearch(problem, fringe):
    fringe.push(problem.initial_state)
    # Get a node in the fringe and expand it if it is not a solution
    while fringe.notEmpty():
        node = fringe.pop()
        if problem.isGoal(node.state):
            return node.solution
        fringe.pushAll(expand(node, problem))
    return FAILURE

def expand(node, problem):
    successors = set()
    # List all neighboring nodes
    for action, result in problem.successor(node.state):
        s = new Node(
            parent=node, action=action, state=result, depth=node.dept+1,
            cost=node.cost + problem.pathCost(node, s, action)
        )
        successors.add(s)
    return successors
```

2.1.3 Evaluation

Completeness

Completeness if the strategy is guaranteed to find a solution (when exists).

Time complexity

Time complexity time needed to complete the search.

Space complexity

Space complexity memory needed to complete the search.

Optimality

Optimality if the strategy finds the best solution (when more solutions are possible).

2.2 Non-informed search

2.2.1 Breadth-first search (BFS)

Breadth-first search

Always expands the less deep node. The fringe is implemented as a queue (FIFO).

Completeness	Yes
Optimality	Only with uniform cost (i.e. all edges have same cost)
Time and space complexity	$O(b^d)$, where the solution depth is d and the branching factor is b (i.e. each non-leaf node has b children)

The exponential space complexity makes BFS impractical for large problems.

2.2.2 Uniform-cost search

Uniform-cost search

Same as BFS, but always expands the node with the lowest cumulative cost.

Completeness	Yes
Optimality	Yes
Time and space complexity	$O(b^d)$, with solution depth d and branching factor b

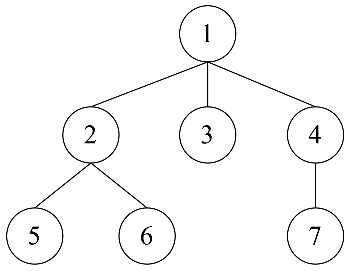


Figure 2.2: BFS visit order

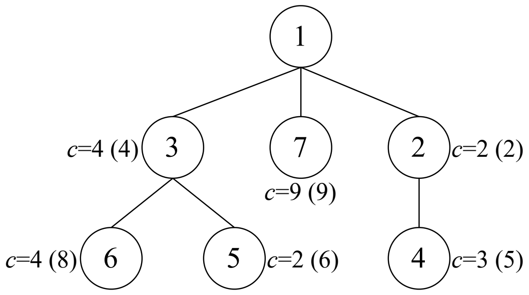


Figure 2.3: Uniform-cost search visit order. (n) is the cumulative cost

2.2.3 Depth-first search (DFS)

Always expands the deepest node. The fringe is implemented as a stack (LIFO). Depth-first search

Completeness	No (loops)
Optimality	No
Time complexity	$O(b^m)$, with maximum depth m and branching factor b
Space complexity	$O(b \cdot m)$, with maximum depth m and branching factor b

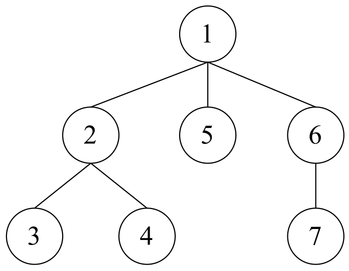


Figure 2.4: DFS visit order

2.2.4 Depth-limited search

Same as DFS, but introduces a maximum depth. A node at the maximum depth will not be explored further. Depth-limited search
This allows to avoid infinite branches (i.e. loops).

2.2.5 Iterative deepening

Iterative deepening

Rus a depth-limited search by trying all possible depth limits. It is important to note that each iteration is executed from scratch (i.e. a new execution of depth-limited search).

Algorithm 2 Iterative deepening

```
def iterativeDeepening(G):
    for c in range(G.max_depth):
        sol = depthLimitedSearch(G, c)
        if sol is not FAILURE:
            return sol
    return FAILURE
```

Both advantages of DFS and BFS are combined.

Completeness	Yes
Optimality	Only with uniform cost
Time complexity	$O(b^d)$, with solution depth d and branching factor b
Space complexity	$O(b \cdot d)$, with solution depth d and branching factor b

2.3 Informed search

Informed search uses evaluation functions (heuristics) to reduce the search space and estimate the effort needed to reach the final goal. Informed search

2.3.1 Best-first search

Uses heuristics to compute the desirability of the nodes (i.e. how close they are to the goal). The fringe is ordered according the estimated scores. Best-first seacrh

Greedy search / Hill climbing The heuristic only evaluates nodes individually and does not consider the path to the root (i.e. expands the node that currently seems closer to the goal). Greedy search / Hill climbing

Completeness	No (loops)
Optimality	No
Time and space complexity	$O(b^d)$, with solution depth d and branching factor b

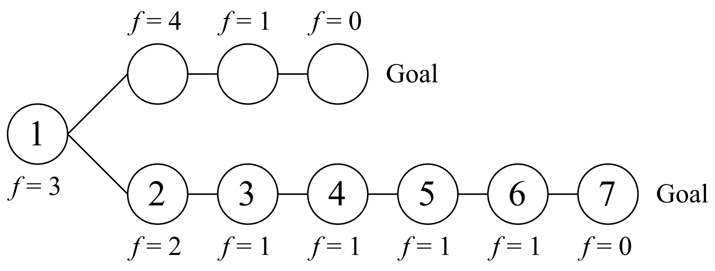


Figure 2.5: Hill climbing visit order

A* The heuristic also considers the cumulative cost needed to reach a node from the root. The score associated to a node n is:

A*

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

where g is the depth of the node and h' is the heuristic that computes the distance to the goal.

Optimistic/Feasible heuristic Given $t(n)$ that computes the true distance of a node n to the goal. An heuristic $h'(n)$ is optimistic (i.e. feasible) if:

Optimistic/Feasible heuristic

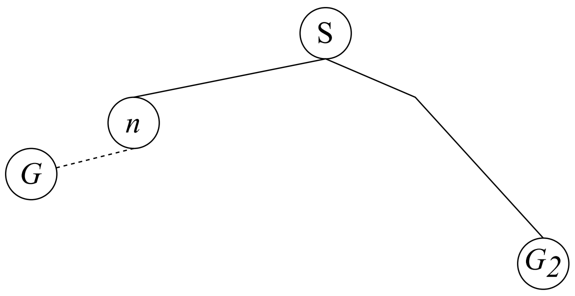
$$h'(n) \leq t(n)$$

In other words, h' is optimistic if it always underestimates the distance to the goal.

Theorem 2.3.1. If the heuristic used by A* is optimistic $\Rightarrow$ A* is optimal

Proof. Consider a scenario where the queue contains:

- A node n whose child is the optimal solution
- A sub-optimal solution G_2



We want to prove that A* will always expand n.

Given an optimistic heuristic $f(n) = g(n) + h'(n)$ and the true distance of a node n to the goal $t(n)$, we have that:

$$f(G_2) = g(G_2) + h'(G_2) = g(G_2), \text{ as } G_2 \text{ is a solution: } h'(G_2) = 0$$
$$f(G) = g(G) + h'(G) = g(G), \text{ as } G \text{ is a solution: } h'(G) = 0$$

Moreover, $g(G_2) > g(G)$ as G_2 is suboptimal. Therefore, $\boldsymbol{f(G_2) > f(G)}$.

Furthermore, as h' is feasible, we have that:

$$\begin{aligned} h'(n) \leq t(n) &\iff g(n) + h'(n) \leq g(n) + t(n) = g(G) = f(G) \\ &\iff \boldsymbol{f(n) \leq f(G)} \end{aligned}$$

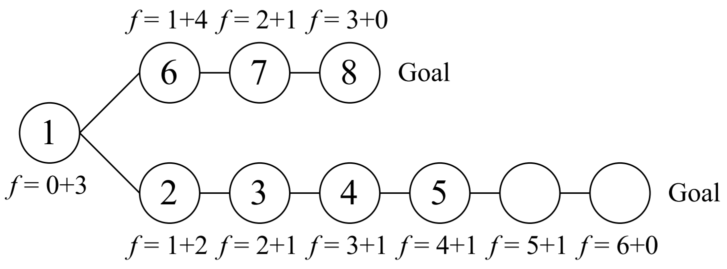
In the end, we have that $f(G_2) > f(G) \geq f(n)$. So we can conclude that A* will never expand G_2 as:

$$f(G_2) > f(n)$$

□

Completeness	Yes
Optimality	Only if the heuristic is optimistic
Time and space complexity	$O(b^d)$, with solution depth d and branching factor b

In generally, it is better to use heuristics with large values (i.e. heuristics that don't underestimate too much).



$f = 1+4$ $f = 2+1$ $f = 3+0$
6
7
8
Goal
1
$f = 0+3$
2
3
4
5
Goal
$f = 1+2$ $f = 2+1$ $f = 3+1$ $f = 4+1$ $f = 5+1$ $f = 6+0$

Figure 2.6: A* visit order

2.4 Graph search

Graph search

Differently from a tree search, searching in a graph requires to keep track of the explored nodes.

Algorithm 3 Graph search

```
def graphSearch(problem, fringe):
    closed = set()
    fringe.push(problem.initial_state)
    # Get a node in the fringe and
    # expand it if it is not a solution and is not closed
    while fringe.notEmpty():
        node = fringe.pop()
        if problem.isGoal(node.state):
            return node.solution
        if node.state not in closed:
            closed.add(node.state)
            fringe.pushAll(expand(node, problem))
    return FAILURE
```

2.4.1 A* with graphs

A* with graphs

The algorithm keeps track of closed and open nodes. The heuristic $g(n)$ evaluates the minimum distance from the root to the node n.

Consistent heuristic (monotone)

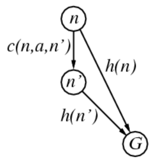
Consistent heuristic (monotone) An heuristic is consistent if for each n, for any successor n' of n (i.e. nodes reachable from n by making an action) holds that:

$$\begin{cases} h(n) = 0 & \text{if the corresponding status is the goal} \\ h(n) \leq c(n, a, n') + h(n') & \text{otherwise} \end{cases}$$

where $c(n, a, n')$ is the cost to reach n' from n by taking the action a.

In other words, f never decreases along a path. In fact:

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$



Theorem 2.4.1. If h is a consistent heuristic, A* on graphs is optimal.