Foundations of Artificial Intelligence

Riccardo Salvalaggio

19th of April, 2021

Contents

1 Introduction

Artificial Intelligence is the attempt to make computers more ”intelligent” to better understand human intelligence.

2 Rational Agents

It is a model that perceive the environment through sensors and act through actuators. In order to evaluate their performance use performance measure (e.g. vacuum -¿ level of cleanliness etc.) even if optimal behaviour is often unattainable because it is quite impossible to reach the goal in every aspect.
Omniscent if it knows the effects of its actions.
Rational agent behaves according to its percepts and knowledge and attempts to maximize the expected performance.
Ideal: for each possible percept sequence, selects an action that is expected to maximize its performance measure.

2.1 Structure of Rational Agents

The mapping is realised through an agent program executed on an Architecture which also provides an interface to the environment(percepts, actions)
Agent = Architecture + Program

2.2 Classes of agents

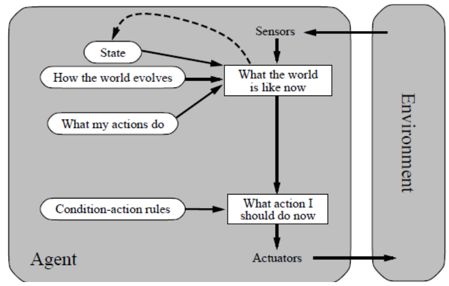
2.2.1 Table-Driven (the simplest)

```
function TABLE-DRIVEN-AGENT(percept) returns an action
  persistent: percepts, a sequence, initially empty
              table, a table of actions, indexed by percept sequences, initially fully specified

  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action
```

Problem: need a huge table to fulfill all the possible perceptions.

2.2.2 Interpretative Reflex

```
function SIMPLE-REFLEX-AGENT(percept) returns an action
  persistent: rules, a set of condition–action rules

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```



Interpretation of the input, matching to a rule to extract an action.

2.2.3 Model-Based Reflex

```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
  persistent: state, the agent's current conception of the world state
              model, a description of how the next state depends on current state and action
              rules, a set of condition–action rules
              action, the most recent action, initially none

  state ← UPDATE-STATE(state, action, percept, model)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

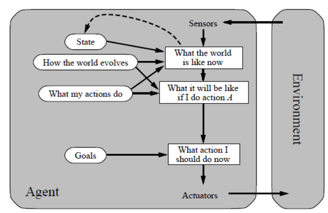


Figure 1: Goal-based

Introduction of a utlity function that maps a state onto a real number in order to compute the best action to do and to weigh the importance of competing goals.

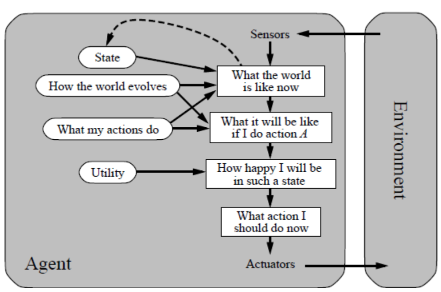
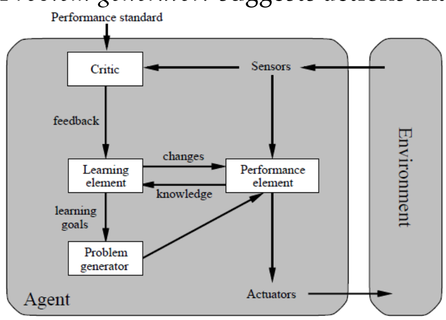


Figure 2: Utility-based

2.2.4 Learning agents

Agents that improve over time starting from an empty knowledge and unknown environments.
Components:
Learning element: responsible for making improvements.
Performance element: select external actions.
Critic: determines performance of the agent.
Problem generator: suggests actions that lead to informative experiences.



2.3 Types of environments

Accessible vs. inaccessible, Deterministic vs. stochastic, Episodic vs. sequential, static vs. dynamic, discrete vs. continuous, single vs. multi agent.

3 Solving Poblems by Searching

3.1 Problem-solving agents

Formulation: *problem* as a *state-space* and *goal* as a *particular condition on states*

Given: *initial state*

Goal: To reach the specified goal (a state) through the *execution of appropriate actions*

- Properties: Fully-observable, Deterministic/static env., discrete states, single-agent.

3.2 Problem Formulation

Goal formulation, definition of: State space, actions, problem type, search and execution costs.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
     goal ← FORMULATE-GOAL(state)
     problem ← FORMULATE-PROBLEM(state, goal)
     seq ← SEARCH(problem)
     if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

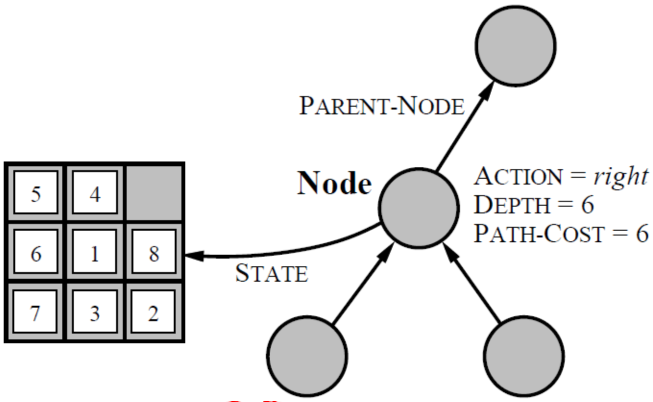
Figure 3: Simple Problem-solving Agent

3.3 Problem Types

Based on knowledge of States and Actions: Observability, completeness of knowledge about world state and actions. (e.g. If the environment is completely observable, the vacuum cleaner always knows where it is and where the dirt is.)
Transition Model: Description of the outcome of an action.
Solution: Path from the initial to a goal state.
Search Costs: Time and storage requirements to and a solution.
Total Costs: Search costs + path costs.
Alternative formulations can influence a lot number of states, e.g. 8 queens problem: Naive - billions of state, Better - 2057 states.
Examples of Real-World Problems: Route planning, shortest path problem, TSP, VLSI Layout, Robot nav., Assembly sequencing.

3.4 Search strategies

E.g.: node expansion, frontier, search strategy, tree-based search, graph-based search.
- Search Tree Data structure:
state, parent, action, path-cost.



function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand ~~the chosen node, adding the resulting~~ nodes to the frontier
      only if not in the frontier or explored set

- Criteria for Search Strategies: ***Completeness, Time complexity, Space Complexity, Optimality***.

3.4.1 Uninformed or blind searches

- Breadth-First Search:
Nodes are expanded in the order they were produced (first siblings, then children) (frontier = FIFO queue). Completeness is obvious, the solution is optimal. **Time complexity:** Let b be the maximal branching factor and d the depth of a solution path. Then the maximal number of nodes expanded is = O(b at d). **Space Complexity: O(b at d)**

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

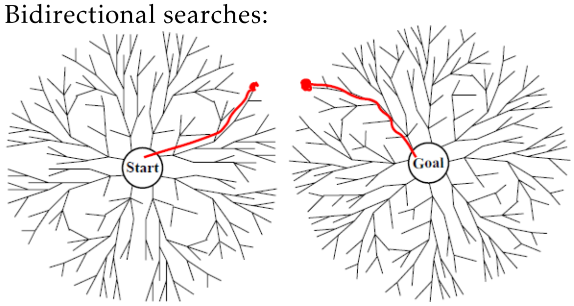
- Uniform-Cost Search:
If step costs are different, uniform cost is better. It expands node with lowest path costs *g(n)*. It uses a Priority queue. Always finds the cheapest solution, given that g(successor(n)) >= g(n) for all n.
- Depth-First Search:
Always expands an unexpanded node at the greatest depth (frontier < – a LIFO queue, first children, then siblings). Usually implemented recursively.
Generally, optimal is not guaranteed. Completeness only for graph-based search. **Time complexity:** in graph-based is bounded by the space, so it can be infinite, in tree-based: O(b at m) (m max length of a path). **Space Complexity:** tree-based: O(b*m), graph-based: worst-case, all states need to be stored. (no better than breadth-first).

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
     cutoff_occurred? ← false
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        result ← RECURSIVE-DLS(child, problem, limit − 1)
        if result = cutoff then cutoff_occurred? ← true
        else if result ≠ failure then return result
     if cutoff_occurred? then return cutoff else return failure
```

- Iterative Deepening Search:
Like depth-limited search and in every iteration increase search depth by one. Combines depth and breadth-first. Optimal and complete like breadth-first, but requires much less memory: O(b*d).

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to ∞ do
     result ← DEPTH-LIMITED-SEARCH(problem, depth)
     if result ≠ cutoff then return result
```

Iterative deepening in general is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.
For small space it is worse than breadth-fist.
- Bidirectional searches:



As long as forward and backward searches are symmetric, search times of O(2*b at d/2) = O(b at d/2) can be obtained. The operators are not always reversible, there must be an efficient way to check if a new node already appears in the search tree of the other half of the search.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

4 Informed Search Methods

- Uninformed: rigid procedure with no knowledge of the cost of a given node to the goal.
- Informed: knowledge of the worth of expanding a node n is given in the form of an evaluation function f(n) which assigns a real number to each node. Mostly, f(n) includes as a component a heuristic function h(n), which estimates the costs of the cheapest path from n to the goal.
- Best-first: informed that expands with the best f-value.

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding solution
      expand the chosen node, adding the resulting nodes to the frontier
```

Instance of tree-search algorithm in which frontier is a priority queue. When f is always correct, we don't need to search.

4.1 Greedy Search

h(n) = estimated path-costs from n to the goal. A best-first search using h(n) (heuristic function) as evaluation function is called greedy search.

4.2 Heuristics

Heuristics are fast but in certain situations incomplete methods for problem-solving, they improve the search in the average-case and the time complexity. In general, not optimal and incomplete; graph-search version is complete only in finite spaces.

4.3 A* and IDA*

A* combines greedy search with the uniform-cost search: always expand node with lowest f(n) = g(n) (actual cost from start to n) + h(n) (estimated cost to goal/optimistic estimate of the costs). A new h is *admissible* iff: h(n) <= h*(n).

4.3.1 Optimality of A*

Claim: The first solution found has the minimum path cost.
Proof: Suppose there exists a goal node G with optimal path cost f*, but A* has first found another node G2 with g(G2) ¿ f*. Let n be a node on the path from the start to G that has not yet been expanded. Since h is admissible, we have: f(n)<=f* −− > f(G2)<=f(n) −− > f(G2)<=f* ==> g(G2)<=f* Contraddiciton.
- Completeness: If a solution exists, A* will find it provided that (1) every node has a finite number of successor nodes, and (2) there exists a positive constant δ ¿ 0 such that every step has at least cost δ.
- Complexity: In general, still exponential in the path length of the solution (space, time), it depends on the choice of Heuristic used.

4.3.2 Graph- vs. Tree-search

For the graph-based variant, either needs to consider re-opening nodes from the explored set, when a better estimate becomes known, or needs to require stronger restrictions on the heuristic estimate: it needs to be consistent (iff for all actions a leading from s to s': h(s) - h(s') <= c(a), where c(a) denotes the cost of action a). Consistency implies admissibility, A* can still be applied if heuristic is not consistent but optimality is lost.

4.3.3 Variants of A*

In general suffers from exponential memory growth. - Iterative-deepening A*: f-costs are used to define the cut-off (IDA*).
- Recursive Best First Search (RBFS): introduces a variable *f-limit* to keep track

of the best alternative path, if the limit is exceeded opt for the alternative path.
- MA* and SMA*.

4.4 Local Search Methods