AI Course Notes - Polimi

Saturday, May 2, 2020 12:44

"This is not an official course material, and it's not guaranteed to be nor complete or error-free. The notes are integrated in part from the book "Artificial Intelligence : A modern approach" and some sentences are taken directly from it. The material for this work is taken also from various notes, books and schemes coming from other students."
Thanks for reading, Gabriele Aquaro

- ## INTELLIGENT AGENTS :

An agent is anything that can be viewed as perceiving its environment through **sensors** and acting upon that environment through **actuators**.

The agent behavior is described by the **agent function** that maps every percept sequence to an action. (Tabulating it is extremely long and resource demanding in most cases, usually "**programmed"**).

F : P* --> A [Agent function F from Perception set to Action set)

Rational Agent :

For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

This is different from an **omniscient agent,** because the performance expected from the rational agent, obtained by the sensors clues, can be different from the real outcome ("known by an omniscient agent").
An agent is able to **Learn** from the environment if after more observation of the environment it gains experience. This is not possible if the environment is known *a priori.*

We define an agent firstly under the heading of the task environment (**PEAS:** Performance, Environment, Actuators, Sensors).

Task Environment categories :

- Single Agent | Multi Agent
- Fully Observable | Partially observable
- Deterministic | Stochastic
- Episodic | Sequential
- Static | Dynamic (Dynamic means the environment changes while the agent is deliberating)
- Discrete | Continuous (Both referred to input and states)
- Known | Unknown (Generally referring to the **"Laws"** of the environment)

Agent Program: Takes the state in input from the sensor and return the action to the actuators.

Type of Agents:

- **Simple reflex** :
  implements a set of IF-Then rules to apply at the current input.
- **Model-based reflex** :
  This agent maintain a sort of internal state that depends on the percept history and reflects some unobserved aspect of the system.
  Has to represent the environment with a model (state).
- **Goal based:**
  Needs the description of a goal-state and try to find a strategy to reach it.
  Search and *Planning* are the subfields of AI devoted to find this actions sequence
- **Utility based:**
  Define an utility function that gives a performance measure. The expected utility of the action outcome is given from the utility function.
  (Goal based can be seen as a Boolean utility function)

Problem formulation:

- **Initial State** : Initial configuration of the environment
- **Actions**: A(s) = {a1, a2, a3, a4…..} | Gives the possible actions applicable in s
- **Result Function** : R(s , a) = s' with s, s' ∈ State-Space and a ∈ Action(s)
- **Goal Test :** G(s) = {0,1} | 1 if the state is a goal state, 0 else way

• SEARCHING:

The research of the solution (Goal State) in the State-Space can follow different techniques .
We need to give some formal definition as first:

- **State:** Represent one possible configuration of the environment in a formal and fixed way

- **Search Tree**: Data structure used to search for a solution, the root node is the initial state, with cost zero. The child of a node are composed by the possible states reachable from the father and the costs of the action up to the node.

- **Node :** Element of the search tree, it's the union of the state, the parent node, the applied action and the cost to reach it.
  A Node in the Search-Tree is a **PATH** in the State-Space Graph.

- **Frontier:** List of nodes that have been generated , but not yet EXPANDED

- **Closed List: (Graph Search)** The expansion only add nodes corresponding to states not already present in one node of the tree. ("Only new states")

- **Completeness:** If a solution exists, a complete search strategy will find it.

- **Optimality:** A optimal search strategy will always find an optimal solution if it exists.

- **Complexity:**
  Spatial : Measured as kept nodes
  Temporal: Measured as generated nodes

- **Branching Factor (b) :** Maximum number of children of each node:
  Maximum cardinality of the return of the action function.

- **Minimum depth (d):** Depth of the shallowest solution (minimum number of actions to reach the goal state)

- **Longest Path (m) :** Length of the longest path in the state space graph, can be infinite with loopy graphs

- **Optimal solution cost (c*):** Minimum cumulative action cost to reach the goal from S_0

- **Minimum action cost (ε) :** Minimum cost of an action

• UNINFORMED SEARCH STRATEGIES:

- **Breadth-first search:**
  The root is expanded first, when a root is generated, it's applied the goal test.
  The frontier is a **FIFO QUEUE** and so the nodes at the a lower depth are expanded first. It guarantees to find a solution if it exists and it's reachable in a finite number of steps. It's not an optimal procedure (except for all unitary/constant action costs).
  This guarantees to find the shallowest solution.
  Spatial complexity : O(b^d)
  Time complexity : O(b^d)
  (If Goal test is applied when the node is EXPANDED the complexity increases to O(b^{d+1})

- **Uniform cost search:**
  Search in the state space expanding the node with the minimum path cost.
  It's complete if the SS has only positive costs, it's optimal. (Dijkstra)

Time complexity : $O(b^{floor(c^*/\varepsilon)})$
Spatial complexity: Equal Time

- **Depth-first search:**
  Search in the State-Space following the LIFO policy, this follows the longest path, trying to go as deep in the tree as possible. This is not complete in a loopy state space using tree search, while with the graph search it will explore the complete graph in the worst case scenario, but it will be complete.
  It's not optimal.
  With graph search*:*
  This technique has a good *spatial complexity*: O(m)
  Time complexity: O(#State)

- **Depth-limited search:**
  Based on depth first strategy, this technique will fix the maximum depth to expand at the beginning (**l**) and will expand following the depth-first strategy:
  It's not complete if l < d (depth of the shallowest solution)
  It's complexity with graph and tree search is
  Time complexity : $O(b^l)$
  Spatial complexity: O(b*l)

- **Iterative deepening search:**
  This technique will fix the maximum depth to expand at each ITERATION, starting from 1 going up until it finds the solution.
  Follows the depth first policy of selection.
  The tree is redone at every iteration in general.
  It's complete and not optimal (like breadth-first finds the shallowest solution)
  Spatial complexity: O(b*d)
  Temporal Complexity: $O(b^d)$

- <u>INFORMED SEARCH STRATEGIES:</u>

These techniques use the additional information about the problem to solve to "lead" the navigation in the search tree.
The main idea is to have an ***evaluation* function f(n)** that given a node returns a Real Number, representing the cost to reach the solution from that node.
As it's impossible to have the correct *evaluation functio*n I most of the cases, we will use an Heuristic function that try to estimate it. (h(n))
So we have that f(n) often depends on h(n).

- **Greedy best-first strategy**:
  This search strategy tries to expand always the node that is closest to the goal, following the h(n) as metric:

  --> *f(n) = h(n)*

  This is complete only using graph search and it's not optimal in general.
  The efficacy of this technique is strictly correlated with the "goodness" of the Heuristic function: if its's perfect, this will follow the perfect path.
  Spatial Complexity: $O(b^m)$
  Time Complexity: $O(b^m)$

- **A* Search:**
  This strategy main idea is to combine the h(n) and the cost to reach the node:

  -> *f(n) = g(n) +h(n)*

  Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have:

  -> *f(n)* = estimated cost of the cheapest solution passing through *n*

  This is identical to the "*Uniform Cost Search"* Except that it uses *h + g* instead of *g.*
  This technique is always complete (with all positive action - costs) and it is guaranteed to be optimal :

Using tree search, if h(n) is **admissible:** $h(n) \leq h^*(n)\ \forall n$

Using graph search, if h(n) is **consistent:** $h(\eta) \leq c(n,n') + h(n') \quad \forall n, n' \mid n' successor\ of\ n$

Notice that ***consistency => admissibility*** and that the consistency property is a *triangular inequality.*

- ## ADVERSARIAL SEARCH:

Based on two player min and max, max is trying to maximize the utility while min is trying to minimize it.

- **Minmax algorithm:**
  The minimax algorithm computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds.

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

  The minmax algorithm always finds the utility of each node and propagate it to the top. This is impractical for the real games.

- **Alpha-Beta Pruning:**
  Given the minmax algorithm problems, we can try to reduce the number of nodes generated/evaluated pruning the game tree in the right place.
  In particular the alpha-beta pruning technique is applied to the minmax tree and returns the same move as minmax would, but prunes away branches that cannot possibly influence the final decision.
  Alpha–beta pruning can be applied to trees of any depth, and I t is often possible to prune entire subtrees rather than just leaves. The general principle is this: consider a node *n* somewhere in the tree, such that Player has a choice of moving to that node. If Player has a better choice *m* either at the parent node of *n* or at any choice point further up, then *n* will never be reached in actual play. So once we have found out enough about *n* (by examining some of its descendants) to reach this conclusion, we can prune it.

- **Montecarlo Tree Search:**
  The Montecarlo tree search strategy exploit the information given by random playout of the game from a selected state to compute the best move. This algorithm has always a ready to use solution, but with more time of execution the solution will converge to the optimal move.
  The Algorithm uses two number for each node and is divided in four stages
  - *N*: Number of simulation executed from a node.
  - *Q*: Sum of the rewards obtained from all the simulation started from a node.
  - *Selection:* During this stage the best node to expand is selected, the algorithm chooses one node between the not expanded ones of the most promising node (*highest Q/N)*
  - *Expansion:* Unless the selected node corresponds to a terminal state, it finds the children of that node and expand the tree.
  - *Simulation:* Plays random moves until a terminal state is found.
  - *Backpropagation:* Propagate the reward of the last state of the simulation to all the upper nodes in the tree.

  After a finite number of cycles (t) or a defined time, the best moves is chosen from the actual state n according to the formula :

$$\frac{Q}{N} + C\sqrt{\frac{\ln t}{N}} \quad where\ t\ is\ the\ number\ of\ cycles\ executed$$

  This procedure allows to balance the tree between the promising nodes and the uncertain ones, avoiding the trap of local maxima. (**optimism in face of uncertainty**)

- ## CONSTRAINTS SATISFACTION PROBLEMS:

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

X is a set of variables, $\{X_1,...,X_n\}$.
D is a set of domains, $\{D_1,...,D_n\}$, 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, $\{v_1,...,v_k\}$ for variable X_i. Each constraint C_i consists of a pair <*scope, relation*>, where *scope* is a tuple of variables that participate in the constraint and *relation* 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 value that satisfy the constraint, or as an abstract relation that support two operations: testing is a tuple is a member of the relation and enumerating the members of the relation.
The simplest kind of CSP involves variable that have **discrete, finite domains.**
Others type of CSP can have **infinite** domains, as for example a set of integers or strings. The problem with this is that is no more possible to enumerate all possible combinations of values, but a **constraint language** must be used.
In real world applications CSP with **continuous** variables are often found. The best known category of continuous-domain CSPs are the **Linear Programming** problems, where constraints are linear inequalities.

- **Constraint graph:**
  The nodes of this graph represent the variables of the problem and the edge the presence of a constraint between two variables.
  This can only be applied to problems that have ***binary constraints*** and variable with a discrete and finite domains

- **Constraints propagation:**
  In CSPs an algorithm can do two things: **search** (choose a new variable to assignment from several possibilities) or do **inference** in the form of **constraint propagation:**
  > *Using the constraints to reduce the number of legal values for a not yet assigned variable, which inn turn can reduce the legal values of another one, and so on.*

  Constraints propagation can be done as a preprocessing step or can be intertwined with search.

- **Backtracking:**
  Adopt a classical Depth First Strategy and at every step check the consistence of the new children, if no child is consistent then backtrack.

- **Backtracking + Forward checking:**
  Build an auxiliary data structure with the *ACTUAL* domains of each variable and every time a new variable is assigned then update the domains of the other basing of the constraints.
  If one domain is empty after updating, backtrack.

- **Heuristic for variable selection:**
  Can adopt a *least values first heuristic*, where the variable with less possible values is chosen first.
  Can use the *Degree heuristic* to choose which variable to assign first (higher degree first).

- **Heuristic for value selection**:
  Least constraining value: choose the value that leaves most freedom to the other variables.

- **Arc consistency:**
  Can be applied only if all the constraints are binary.
  Based on a queue of constraints, proceeds to reduce the domains removing the elements from the queue, stops when the queue is empty or fids an empty domain.
  1) Initialize the queue with all the arcs (constraints) in a random order (considering both the orientation of the arc x_i -> x_2 implies x_2 -> x_1)
  2) Loops until end:
     - Take the first arc from the queue.
     - Check the consistency of the left node (x_1 if the arc is x_1 -> x_2): x_1 is consistent if and only if for each value of his domain there is a value of x_2 that is valid.
     - If consistent then remove it from the node.
     - If not remove from the x_1 domains the non valid values and than reinsert into the queue every arch x_i -> x_1 | not in the queue and x_i != x_2 .
  3) If the queue is empty and there are no empty domains, than the found domains are the legal ones. Unsatisfiable if at least one domain is empty.

- <u>LOGICAL AGENTS:</u>

The objective of logical agents is to infer new information from a KB in the better way possible.
Entailment*:*
Given a KB and a sentence α, we say that the KB entails α, if and only if for every model in which the KB is true, also α is true:

KB |= α .

Logical inference:
Given a sentences α and a Knowledge Base KB, we say that α is derived from KB given the procedure *p,* if it can be derived in a finite number of steps starting from KB using *p:*

KB $\vdash_p$ α .

A procedure is said to be **sound** if it derives only sentences that are entailed by the knowledge base.
A procedure is said to be **complete** if it derives all the sentences that are entailed by the knowledge base.

CNF:
To make inference in a knowledge base, can be useful to have all the formulas in Conjunctive Normal Form (CNF):
"Every formula in propositional logic is equivalent to a formula in the canonical CNF"

A generic formula is written in CNF if it presents only if it is written as a **conjunction of clauses**, where a clause is:
(A) , (A v B), (A v ¬B) …
So a formula in CNF is in the shape of : (A v B) ∧ (¬A v C)

Tips and tricks to CNF :

$A \Leftrightarrow B \equiv \beta \Rightarrow (\mathrm{A} \Rightarrow B) \ \wedge \ (\mathrm{B} \Rightarrow A)$
$(\mathrm{A} \Rightarrow B) \ \equiv \ \neg \mathrm{A} \vee \mathrm{B}$
$\mathrm{A} \vee (\mathrm{B} \wedge \mathrm{C}) \ \equiv (\mathrm{A} \vee \mathrm{C}) \ \wedge \ (\mathrm{A} \vee \mathrm{B})$

Definite clauses:
A clause is definite if it is a single literal or it's an implication whose premises is a atomic literal or a conjunction of literals and whose conclusion is a atomic literal.

Forward chaining:
This kind of procedure need to have all the KB written with **Definite Clauses.**
A forward chaining algorithm starts from the known facts, then triggers all the possible rules whose premises are satisfied, adding their conclusion to the known facts.
(EX. Applying *modus pones*

1. A ∧ B ⇒ C
2 A
3 C
Can derive C has True)

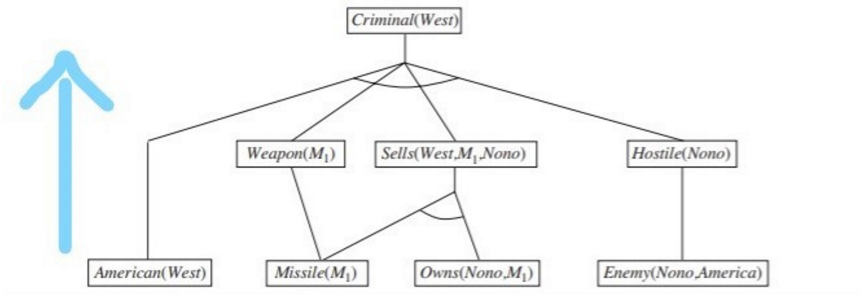

Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

FWC is **complete and sound.**

Backward chaining:
This kind of procedure need to have all the KB written with **Definite Clauses.**
Backward chaining procedures work in the opposite way in respect of *forward chaining,* they start from the "objective" of the inference process, then trigger the rules that have as consequence the already derived facts.

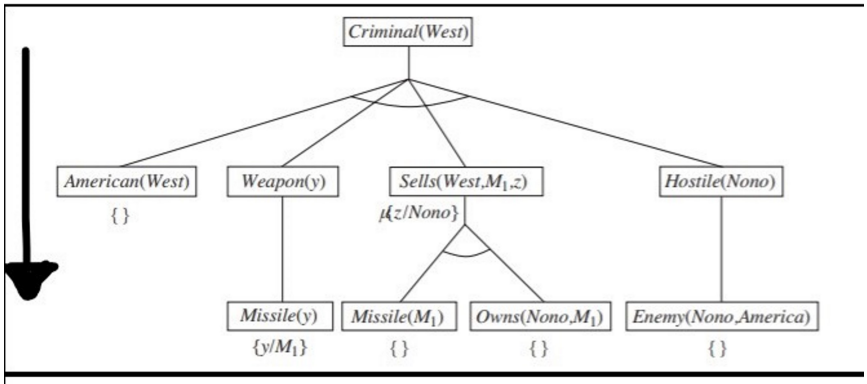

Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove $Criminal(West)$, we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally $Hostile(z)$, z is already bound to $Nono$.

Backward chaining in not complete

Resolution:
Inference procedures based on resolution work by using the principle of **proof by contradiction** (aka: Add at the KB, the negation of the formula you want to infer) .
That is, to show that KB |= α, we show that (KB ∧ ¬α) is unsatisfiable.
The procedure follows these steps :

i. Write the all the KB in CNF
ii. Write all the clauses of the formulas
iii. Found new clauses using the **resolution rule :** two clauses with complementary literals can be simplified (A v B, C v ¬B becomes A v C)
iv. Continue to simplify formulas until the Empty clause is found, but always keep into consideration at least one clause from the original formulas.
v. Stop if the empty clause is found, A, ¬A : can't be resolved ⇒ empty clause ⇒ KB |= α. if no more formulas can be deduced and the empty clause can't be deduced, stop and KB |\= α .

DPLL:
The DPLL algorithm start from a KB in CNF (so a set of clauses) and a sentence α, to establish if KB |= α.

i. Start from adding ¬α to the KB.
ii. Iteratively apply the "*Pure symbol Heuristic*" or "*Unit Clause heuristic*" to the sentences:
Choose one literal according to the chosen heuristic and add it (with one "truth value" T or F) to the Model.
iii. According to the value of the literal added to the model, update the list of clauses.
iv. Loop [2] until the empty clause is found or all the literals are consistently assigned.
v. If the empty clause is found, then KB ∧ ¬α is not satisfiable by any model, and so KB |= α.

Handle FOL :
As known FOL has variables and functions, not only literals. ON a KB written in FOL can be done inference using all the three methods described above using the

- <u>PLANNING:</u>

A **Plan** is a sequence of actions, with the objective of reaching the goal.
The representation for planning problems scales up to problems that could not be handled by Search-Based representation and by Logic-Based.
Planning is based on a factored representation, one in which a state of the world is represented by a collection of variables.
PDDL is the Planning Domain Definition Language that allows to represent a world with variable and a action schema.
Each state is represented as a conjunction of fluents that are ground, functionless atoms.
The **database-semantic** is used: the **closed-world assumption** means that any fluents that are not mentioned are false, and the **unique name assumption** means that two object "A" and "B" are distinct.
The representation of states is carefully designed so that a state can be treated either as a **conjunction of fluents**, which can be manipulated by logical inference, or as a set of fluents, which can be manipulated with **set operations**.
Actions are described by a set of action schemas that implicitly define the ACTIONS(s) and RESULT(s, a) functions needed to do a problem-solving search. PDDL does that by specifying the result of an action in terms of what changes; everything that

stays the same is left unmentioned:

Action(Fly(p, from, to):
PRECOND: At(p, from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
EFFECT: ¬At(p, from) ∧ At(p, to))

The schema consists of the **action name**, a list of all the variables used in the schema, a **precondition** and an **effect**.

We say that action a is **applicable** in state s if the preconditions are satisfied by s.
The **result** of executing action a in state *s* is defined as a state *s'* which is represented by the set of fluents formed by starting with *s*, removing the fluents that appear as negative literals in the action's effects , and adding the fluents that are positive literals in the action's effects.

Solution of a planning problem:

- Forward Planning : Search in the state space applying one classical search algorithm.

- Backward Planning: Search in the goal space, search through sets of relevant states, starting at the set of states representing the goal and using the inverse of the actions to search backward for the initial state.. It is called **relevant-states search** because we only consider actions that are relevant to the goal (or current state).

- SATPlan: Translate a planning problem into a propositional satisfiability one and solve it by finding a model using for example DPLL or resolution.
  How to encode a PDLL in propositional logic :
  - Propositional Symbols : PS^t which means "PS is true at time t"
  - Actions : A^t which means "A is applied at time t" (Effects will be visible at time t+1).
  - Representation at time t :
    Initial state : initial state at time t = 0
    Goal state : all propositional symbols at time t+1
    Precondition axioms : $Action^{t-1} \Rightarrow (PS1^{t-1} \wedge \dots \wedge PSn^{t-1})$ = (precondition of Action)
    Fluent axioms : $Effects^{t+1} \Leftrightarrow Effects^t \vee Action^t$ (where effects are the effects of action)
    Exclusion axiom : $\neg Action\_1^t \vee \neg Action\_2^t, \dots \neg Action\_2^t \vee \neg Action\_n^t$ (one action active at each time)

    (if t = 0 only initial and goal representation)

  - When a model is found the length of the plan is *t* at which the goal is satisfied.