Dipartimento di elettronica e informazione
Politecnico di Milano

Artificial Intelligence 2010-11

4. The state space metamodel

Let us concentrate on *goal-driven agents* (see Section 3). The intuitive idea is that:

- the agent wants to change the environment from its current state to some desired state (the *goal*);
- to do so, the agent will have to perform a sequence of actions on the environment;
- what actions can be performed at every moment *t* depends on the agent's abilities and on the state of the environment at *t*.

Intuitive examples are:

- finding the best way to go from Milan to Vienna and back, to attend a meeting from 2 to 4 pm next Friday;
- finding a good job after getting one's MSc degree in computer science;
- solving a Sudoku puzzle.

Remember that our aim is to design an artificial agent, that is, a software application that can automatically find a way to reach a goal. To do so, we first need to set up a suitable *metamodel* of the agent-environment pair; such a metamodel will consist of a general scheme that tells us how to model specific cases. A metamodel that is widely used in AI is known as the *state space metamodel*.

4.1 State spaces

A *state space* is an ordered quadruple **StateSpace** = $\langle States, Actions, result, cost\rangle$, where:

- *States* = $\{s, s', s'', ...\}$ is a nonempty, countable[1] set of *states* (of the environment);
- *Actions* = $\{a, a', a'', ...\}$ is a nonempty, finite state of *action names* (i.e., symbols denoting actions that can be performed by the agent);
- *result*: $Actions \times States \rightsquigarrow States$ is a partial function[2] that associates a *result* (i.e., a state) to the execution of an action in a state.
- *cost*: $Actions \times States \rightsquigarrow \mathbb{R}$ is a partial function that associates a *cost* (i.e., a real number) to the execution of an action in a state.

Comments:

- Interesting sets of states are either countably infinite or finite but 'large' (which would make it inefficient to represent the whole state space in memory).
- *Actions* must contain at least two different actions, otherwise there would be no issue of choosing rationally what to do.
- Function *result* represents the expected effect *result*(a,s) of executing action a on the environment, when the environment is in state s. Its implementation will have to specify, for every action, the *applicability conditions* of the action (i.e., the states to which the action can be

[1] "Countable" means either finite or denumerable.

[2] A *partial function* is a function whose value may be undefined for certain arguments belonging to the function's domain. A partial function may be also regarded as a *functional binary relation*, that is, as a binary relation that binds every element of its domain to at most one element of its range.

applied) and the *effects* of performing the action (i.e., the state resulting from the execution of the action in a given state). Actions are considered to be *deterministic* (i.e., their results are determined with certainty).

- Function *cost* represents the expected cost $cost(a,s)$ of executing action a on the environment, when the environment is in state s; if a cost is a negative number, then it is interpreted as a gain. In many applications (in particular those involving puzzles) there is no reason to attribute different costs to different action executions, and thus all costs are assumed to be identically equal to 1. Note that the *same action* may have *different costs* when it is executed in *different states* (compare the costs of moving a step forward when you are in the middle of a lawn or on the verge of a precipice...).
- Functions *result* and *cost* are two partial functions that share exactly the same domain of definition; that is, $result(a,s)$ is defined exactly when $cost(a,s)$ is also defined, and vice-versa.
- Given a concrete application, the *state space metamodel* tells us that what has to be done is to: (i), define a suitable set of environment states, *States*; (ii), identify the set of possible actions, *Actions*; define the applicability conditions of each action, and the effects of each action in every state in which it can be executed; and (iv), define the cost of each action in every state in which it can be executed. This done, we will have defined a *state space model* for the concrete application.

4.2 Problems and solutions

A state space provides a context in which we can:

- define *problems*;
- define the concept of a *solution*;
- specify strategies for finding solutions.

Given a state space, **StateSpace** = $\langle States, Actions, result, cost\rangle$, a *problem* in **StateSpace** is an ordered triple **Proplem** = $\langle$**StateSpace**$, s_0, Goals\rangle$, where:

- $s_0 \in States$ is a state, called the *initial state*;
- $Goals \subseteq States$ is a subset of the set of states, whose elements are called *goal states* or *goals*.

A *solution* of **Problem** is a sequence of actions, $\boldsymbol{a} = \langle a_1,...,a_n\rangle$, such that the ordered execution of a_1, ..., a_n starting from the initial state results into a goal state; that is,

$$s_n \in Goals, \qquad \text{where} \quad \begin{aligned} s_n &= result(a_n, s_{n-1}), \\ s_{n-1} &= result(a_{n-1}, s_{n-2}), \\ &... \\ s_1 &= result(a_1, s_0), \end{aligned}$$

or, more concisely,

$$result(a_n, result(a_{n-1}, ..., result(a_1, s_0)...))) \in Goals.$$

The *length* of solution $\boldsymbol{a} = \langle a_1,...,a_n\rangle$ is given by n, the number of actions in the sequence. The *cost* of solution $\boldsymbol{a}$ is the sum of the costs associated to the execution of all actions in the sequence:

$$cost(\boldsymbol{a}, s_0) = \Sigma_i\, cost(a_i, s_{i-1}) \qquad \text{(for } i \text{ from 1 to } n\text{).}$$

When all costs are identically equal to 1, the length and the cost of a solution coincide:

$$cost(\boldsymbol{a}, s_0) = n.$$

Let *Solutions*(**Problem**) be the set of *all* solutions of **Problem**. Solution $\boldsymbol{a^*} \in$ *Solutions*(**Problem**) is said to be *optimal* when no other solution in *Solutions*(**Problem**) has a cost that is lower than the cost of $\boldsymbol{a^*}$.

Note that the optimality of the solution concerns the *primary optimisation problem,* that is, finding a solution with the lowest possible cost. It does not concern the *secondary optimisation problem*, which involves finding a solution with the least possible computational effort.

Finally, note that in general a problem may be unsolvable, that is, it may admit of no solution (i.e., **Problem** may be such that no sequence ***a*** of actions exists that leads from s_0 to *Goals*).

(Example: The 8-puzzle)

4.4 Search strategies

Given **Problem** = ⟨**StateSpace**,s_0,*Goals*⟩, where **StateSpace** = ⟨*States*, *Actions*, *result*, *cost*⟩, we are interested in software applications that are able to produce a solution ***a*** ∈ *Solutions*(**Problem**). Typically, solutions are built by trial-and-error strategies, also known as *search strategies*.

Search *strategies* should not be confused with search *algorithms*: while any strategy has to be represented as an algorithm before it can be programmed on a computer, the *same* strategy may be represented by *different* algorithms (for example, by an iterative algorithm and by a recursive one). Such strategies can be evaluated according to four basic dimensions:

- *Completeness*. A search strategy Σ is *complete* if and only if: every time that **Problem** admits of at least one solution, strategy Σ will find a solution in a finite number of computation steps.
- *Optimality*. A search strategy Σ is *optimal* if and only if: every time that strategy Σ finds a solution of **Problem**, this is an optimal solution. (Note that optimality refers to the *optimality of the solution*, not to the *optimality of the process of finding the solution*!)
- *Time complexity*. The *time complexity* of a strategy is a measure of the number of computation steps required to find a solution, expressed as a function of some parameter of **Problem**.
- *Space complexity*. The *space complexity* of a strategy is a measure of the size of computer memory required to find a solution, expressed as a function of some parameter of **Problem**.