Notas do Curso

Patrick Oliveira

Curso ministrado pelo Prof. Denis Fantinato, 3Q19. As notas envolvem anotações de aula e recortes de livros e artigos.

8 de Outubro de 2019

Different definitions and approaches to AI.

1. **Thinking Humanly**: "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning..."(Bellman, 1978)
2. **Thinking Rationally**: "The study of the computations that make it possible to perceive, reason, and act."(Winston, 1992)
3. **Acting Humanly**: "The art of creating machines that perform functions that require intelligence when performed by people."(Kurzweil, 1990)
4. **Acting Rationally**: "Computational Intelligence is the study of the design of intelligent agents."(Poole *et al.*, 1998)

1 Intelligent Agents

An *agent* is anything that can be viewed as perceiveing its *environment* through *sensors* and acting upon that environment through *actuators*. We use the term *percept* to refer to the agent's perceptual inputs at any given instant. An agent's *percept sequence* is the complete history of everything the agent has ever perceived. Mathematically speaking, we say that an agent's behavior is described by the *agent function* that maps any given percept sequence to an action.

The correct action is decided based on its consequences. When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives. This sequence causes the environment to go through a sequence of states. If the sequence is desirable, then the agent has performed well. This notion of desirability is captured by a *performance measure* that evaluates any given sequence of environment variables.

What is rational at any given time depends on four things:

- The performance measure that defines the criterion of success.
- The agent's prior knowledge of the environment.
- The actions that the agent can perform.
- The agent's percept sequence to date.

This leads to a definition of 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.*

2 Solving Problems By Searching

A **problem** can be defined formally by five components:

- The **initial state** that the agent starts in, e.g. *In(**state**)*.
- A description of the possible **actions** available to the agent. Given a particular stage *s*, *ACTIONS(s)* returns the set of actions that can be excecuted in *s*. We say that each of these actions is **applicable** in *s*.

- A description of what each action does; the formal name for this is the **transition model**, specified by a function *RESULT(s, a)* that returns the state that results from doing action *a* in state *a*. We also use the term **successor** to refer to any state reachable from a given state by a single action. Together, the initial state, actions, and transition model implicitly define a **state space** of the problem. The state space forms a directed network in which the nodes are states and links between nodes are actions.
- The **goal test**, which determines wheter a given state is a goal state.
- A **path cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.

A **solution** to a problem is an action sequence that leads from the initial state to a goal state. An **optimal solution** has the lowest path cost among all solutions.

3 Problem Solving By Searching

A **problem-solving agent** use **atomic** representations, states of the world that are considered as wholes, with no internal structure visible to the problem-solving algorithm. Goal-based agents that use more advanced factored or structured representations are usually called planning agents.

"Intelligent agents are supposed to maximize their performance measure.- *What if an agent is composed by a different set of performance measures, competing to be minimized? How the agent would behave? The decision process would be overly complex and probably impossible to solve "analytically". We, humans, also cannot solve all the decision processes that we commit ourselves to. The constraint is time. At the end of a given time period, we make the decision that seens better under that circunstance.*

Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider. Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving.

We will consider a goal to be a set of world states - exactly those states in which the goal is satisfied. The agent's task is to find out how to act so that it reaches a goal state. Problem formulation is the proess of deciding what actions and states to consider, given a goal.

There can be an agent that learns the map of the environment where it is in? The map can be about the set of possible world states.

If the agent does not have additional information about the problem that it is trying to solve, i.e., if the environment is unknown, then it is has no choice but to try one of the actions at random. But the agent can have a map; the point of a map is to provide the agent with inormation about the states it might get itself into and the actions it can take. The agent can use this information to consider subsequente stages. In general, an agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value.

The environment can be

- **observable:** The agent always knows the current state.
- **discrete:** so at any given state there are only finitely many actions to choose from. *What would be the mathematical consequences if we consider an enumerable set of possible actions? Or a uncountable set of possible actions? There can be a logical formulation of a decision process? Can i prove theorems about that?*
- **known:** the agent knows which states are reached by each action.
- **deterministic:** each action has exactly one outcome.

Under these assumptions, the solution to any problem is a fixed sequence of actions. The process of looking for a sequence of actions that reaches the goal is called a **search**. A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase. While the agent is executing the solution sequence it ignores its percepts when choosing an action because it knows in advance what they will be. Control theorists call this an open-loop system, because ignoring the percepts breaks the loop between agent and environment.

3.0.1 Well-defined problems and solutions

A problem can be defined formally by five components:

- The **initial state** that the agent starts in.

- A description of the possible **actions** available to the agent. Given a particular state s, Actions(s) returns the set of actions that can be executed in s. We say that each of these actions is applicable in s.
- A description of what each action does; the formal name for this is the **transition model**, specified by a function Result(s, a) that returns the state that results from doing action a in state s. Together, the initial state, actions, and transition model implicitly define the **state space** of the problem - the set of all states reachable from the initial state by any sequence of actions. The state space forms a directed network or graph in which the nodes are states and the links between nodes are actions. A **path** in the state space is a sequence of states connected by a sequence of actions.
- The **goal test**, which determines whether a given state is a goal state. SOmetimes the goal is specified by an abstract property rather than an explicitly enumerated set of states.
- A **path cost** function that assigns a numeric cost to each path.

3.1 Searching for Solutions

A solution is an action sequence, so search algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state form a **search tree** with the initial state at the root. The branches are actions and the **nodes** correspond to states in the state space of the problem.

The first step is to test whether this is a goal stage. Then we need to consider taking various actions. We do this by expanding the current state; that is, applying each legal action to the current state, thereby generating a new set of states. In this case, we add three branches from the parent node, leading to new childs nodes. The set of all leaf nodes avaialble for expansion at any given point is called the **frontier**.

The frontier separates the state-space graph into the explored region and the unexplored region, so that every path from the initial state to an unexplored state has to pass through a state in the frontier. As every step moves a state from the frontier, we see that the algorithm is systematically examining the states in the state space, one by one, until it finds a solution.

The process of expanding ndoes on the frontier continues until either solution is found or there are no more states to expand. Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next - the so-called search strategy.

A repeated state in the search tree is generated by a loopy path. Loopy paths means that the complete search tree is infinite because there is no limit to how often one can traverse a loop. Loopy paths are a special case of the more general concept of redundant paths, which exist whenever there is more than one way to get from one state to another. In some cases, it is possible to define the problem itself so as to eliminate redundant paths. In other cases, redundant paths are unavoidable. This includes all problems where the actions are reversible.

The way to avoid exploring redundant paths is to remember where one has been. To do this, we augment the Tree-Search algorithm with a data structure called the explored set (also known as the closed list), which remembers every expanded node.

3.1.1 Infrastructure for search algorithms

For each node n of the tree, we have a structure that contains four components:

- **n.State.**
- **n.Parent.**
- **n.Action.**
- **n.Path-cost.**

The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy. The appropriate data structure for this is a queue.

3.1.2 Measuring problem-solving performance

We can evaluate an algorithm's performance in four ways:

- Completeness: Is the algorithm guaranteed to find a solution when there is one?
- Optimality: Does the strategy find the optimal solution?
- How long does it take to find a solution?

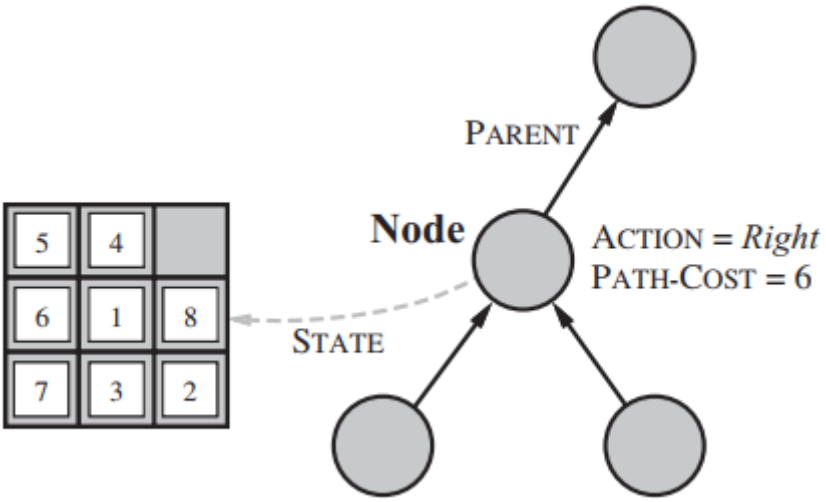

Figura 1: Nodes are the data structures from which the search tree is constructed.

- How much memory is needed to perform the search?

In AI, the graph is often represented implicitly by the initial state, actions, and transition model and is frequently infinite. For these reasons, complexity is expressed in terms of three quantities: b, the branching factor or maximum number of successors of any node; d, the depth of the shallowest goal node. Time is often measured in terms of the number of nodes generated during the search, and space in terms of the maximum number of nodes stored in memory.

To assess the effectiveness of a search algorithm, we can consider just the search cost, or we can use the total cost, which combines the search cost and the path cost of the solution found.

3.2 Uninformed Search Strategies

3.2.1 Breadth-first search

Breadth-first search is an instance of the general graph-search algorithm in which the shallowest unexpanded node is chosen for expansion. This is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. There is one slight tweak on the general graph-search algorithm, which is that the goal test is applied to each node when it is generated rather than when it is selected for expansion. The algorithm is complete, if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes. Now, the shallowest goal node is not necessarily the optimal one.

Listing 1: Breadth-first search on a graph

```
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)
        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)
```

3.2.2 Uniform-cost search

When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step-cost function. Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost g(n). This is done by storing the frontier as a priority queue ordered by g. The goal test is applied to a node when it is selected for expansion, to avoid selecting a solution that was generated first but that is suboptimal.

Listing 2: Uniform-cost search on a graph.

```
function Uniform-Cost-Search(problem) returns a solution, or failure

    node = a node with "State = problem.Initial.State, Path-Cost = 0"
    frontier = a priority queue ordered by Path-Cost, with node as the only element
    explored = an empty set
    loop do
        if Empty?(frontier) then return failure
        node = Pop(frontier)
        if problem.Goal-Test(node.State) then return Solution(node)
        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
                frontier = Insert(child, frontier)
            else if child.State is in frontier with higher Path-Cost then
                replace that frontier node with child
```

Uniform-cost search is optimal in general. First, we observe that whenever uniform-cost search selects a node n for expansion, the optimal path to that node has been found. Then, because step costs are nonnegative, paths never get shorter as nodes are added. These two facts together imply that uniform-cost search expands nodes in order of their optimal path cost. Hence, the first goal node selected for expansion must be the optimal solution.

3.2.3 Depth-first search

Depth-first search always expands the deepest node in the current frontier of the search tree. The search proceeds immediately to the deepets level of the search tree, where the nodes have no successors. It uses a LIFO queue.

The properties of depth-first search depend strongly on whether the graph-search or tree-search version is used. The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces because it will eventually expand every node. The tree-search version, on the other hand, is not complete. Depth-first tree search can be modified at no extra memory cost so that it checks new states against those on the path from the root to the current node; this avoids infinite loops in finite state spaces but does not avoid the proliferation of redundant paths.

3.2.4 Depth-limited search

The embarassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit l. This approach is called depth-limited search. The depth limit solves the infinite-path problem. Unfortunately, it also introduces an additional source of incompleteness if we choose $l < d$. Depth-limited search will also be nonoptimal if we choose $l > d$. Depth limits can be based on knowledge of the problem.

```
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)
```

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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figura 2: Evaluation of tree-search strategies.

```
        if result = cutoff then cutoff_occurred? = true
        else if result != failure then return result
    if cutoff_occurred? then return cutoff else return failure
```

3.2.5 Iterative deepening depth-first search

Iteratie deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit. It does this by gradually increasing the limit until a goal is found.

Iterative deepening search may seem wasteful because states are generated multiple times. It turns out this is not too costly. The reason is taht in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times.

```
function Iterative-Deepening-Search(problem) returns a solution, or failure
    For depth = 0 to inf do
        result = Depth-Limited-Search(problem, depth)
        if result != cutoff then result result
```

It's possible to use a hybrid approach that runs breadth-first search until almost all the avaialble memory is consumed, and then runs iterative deepening from all the nodes in the frontier. In general, iterative deepening is the preferred uninformed search method when the search space is alrge and the depth of the solution is not known.

3.2.6 Bidirectional search

The idea behind bidirectional search is to run two simultaneous searches - one forward from the initial state and the other backward from the goal - hoping that the two searches meet in the middle. The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d.

Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found. The check can be done when each node is generated or selected for expansion and, with a hash table, will take constant time. How to search backward? Let the predecessors of a state x be all those states that have x as a sucessor. Bidirectional search requires a method for computing predecessors. When all the actions in the state are reversible, the predecessors of x are just its successors. Other cases may not be like that.

3.3 Inform