AI Lab - Tutorial

Alessandro Farinelli
Thanks to Davide Corsi, Luca Marzari and Celeste Veronese for help with slides and code

University of Verona
Department of Computer Science

Marzo 18, 2024

Assignments

- Your assignments for this lesson can be found at: *uninf_search/uninf_search_1_problem.ipynb*. You will be required to implement some Uninformed Search algorithms
- In the following you can find pseudocodes for such algorithms

Uninformed Search: tree and graph search versions

```
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
```

Node data structure

Search algorithms require a data structure to keep track of the search tree. A *Node* in the tree is represented by a data structure with three components:

Node(state, parent, pathcost)

- state: the state to which the node corresponds;
- parent: the node in the tree that generated this node;
- pathcost: the total cost of the path from the initial state to this node
- depth: the depth of the node in the search tree. You do not need to initialize this, as this is automatically set by the constructor.

Breadth-First Search (BFS): graph search version

Require: $problem$
Ensure: $solution$

$node \leftarrow$ a node with $\textsc{State} = problem.\textsc{Initial-State}, \textsc{Path-Cost} = 0$
if $problem.\textsc{Goal-Test}(node.\textsc{State})$ **then return** $\textsc{Solution}(node)$
$frontier \leftarrow \textsc{Node-Queue}$
$explored \leftarrow \emptyset$
while not $\textsc{Is-Empty}(frontier)$ **do**
    $node \leftarrow \textsc{Remove}(frontier)$ ▷ Remove last node
    $explored \leftarrow explored \cup node.STATE$
    for each $action$ **in** $problem.\textsc{Actions}(node.\textsc{State})$ **do**
        $child \leftarrow \textsc{Child-Node}(problem, node, action)$
        if $child.\textsc{State}$ **not in** $explored$ **or** $frontier$ **then**
            if $problem.\textsc{Goal-Test}(child.\textsc{State})$ **then return** $\textsc{Solution}(child)$
            $frontier \leftarrow \textsc{Insert}(child)$
return $\textsc{Failure}$

Iterative Deepening Search (IDS): tree search version

```
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
```

```
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
```