

# problem solving by search

Tomas Svoboda  
BE5B33KUI, 2017-02-27

# tree-search

**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

# tree vs graph search

**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*

# child-node

**function** CHILD-NODE(*problem, parent, action*) **returns** a node  
**return** a node with

STATE = *problem.RESULT(parent.STATE, action)*,

PARENT = *parent*, ACTION = *action*,

PATH-COST = *parent.PATH-COST + problem.STEP-COST(parent.STATE, action)*

# breadth-first-search

**function** BREADTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure

*node*  $\leftarrow$  a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0

**if** *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

*frontier*  $\leftarrow$  a FIFO queue with *node* as the only element

*explored*  $\leftarrow$  an empty set

**loop do**

**if** EMPTY?(*frontier*) **then return** failure

*node*  $\leftarrow$  POP(*frontier*) /\* chooses the shallowest node in *frontier* \*/

add *node*.STATE to *explored*

**for each** *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

*child*  $\leftarrow$  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*  $\leftarrow$  INSERT(*child*, *frontier*)

# uniform-cost search

**function** UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

*node*  $\leftarrow$  a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0

*frontier*  $\leftarrow$  a priority queue ordered by PATH-COST, with *node* as the only element

*explored*  $\leftarrow$  an empty set

**loop do**

**if** EMPTY?(*frontier*) **then return** failure

*node*  $\leftarrow$  POP(*frontier*) /\* chooses the lowest-cost node in *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*  $\leftarrow$  CHILD-NODE(*problem*, *node*, *action*)

**if** *child*.STATE is not in *explored* or *frontier* **then**

*frontier*  $\leftarrow$  INSERT(*child*, *frontier*)

**else if** *child*.STATE is in *frontier* with higher PATH-COST **then**

      replace that *frontier* node with *child*

# depth-limited search

**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

**function** ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure  
  **for** *depth* = 0 **to**  $\infty$  **do**  
    *result*  $\leftarrow$  DEPTH-LIMITED-SEARCH(*problem*, *depth*)  
    **if** *result*  $\neq$  cutoff **then return** *result*