CS 316: Search

Stefan D. Bruda

Winter 2023

Problem solving agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT(p) returns an action
inputs: p, a percept
static: s, an action sequence, initially empty
        state, some description of the current world state
        g, a goal, initially null
        problem, a problem formulation

state ← UPDATE-STATE(state, p)
if s is empty then
   g ← FORMULATE-GOAL(state)
   problem ← FORMULATE-PROBLEM(state, g)
   s ← SEARCH(problem)
action ← RECOMMENDATION(s, state)
s ← REMAINDER(s, state)
return action
```


Goal formulation

```
function SIMPLE-PROBLEM-SOLVING-AGENT(p) returns an action
inputs: p, a percept
static: s, an action sequence, initially empty
        state, some description of the current world state
        g, a goal, initially null
        problem, a problem formulation

state ← UPDATE-STATE(state, p)
if s is empty then
   g ← FORMULATE-GOAL(state)
   problem ← FORMULATE-PROBLEM(state, g)
   s ← SEARCH(problem)
action ← RECOMMENDATION(s, state)
s ← REMAINDER(s, state)
return action
```

- A goal is a set of world states (explicit or implicit)

Problem formulation

```
function SIMPLE-PROBLEM-SOLVING-AGENT(p) returns an action
inputs: p, a percept
static: s, an action sequence, initially empty
        state, some description of the current world state
        g, a goal, initially null
        problem, a problem formulation

state ← UPDATE-STATE(state, p)
if s is empty then
   g ← FORMULATE-GOAL(state)
   problem ← FORMULATE-PROBLEM(state, g)
   s ← SEARCH(problem)
action ← RECOMMENDATION(s, state)
s ← REMAINDER(s, state)
return action
```

- Decide the structure (and granularity) of states and what are the possible (elementary) actions

Problem formulation (cont'd)

Environment	Problem type
deterministic, accessible	single-state problem
deterministic, inaccessible	multiple-state problem
nondeterministic, inaccessible	contingency problem
unknown state space	exploration/online problem

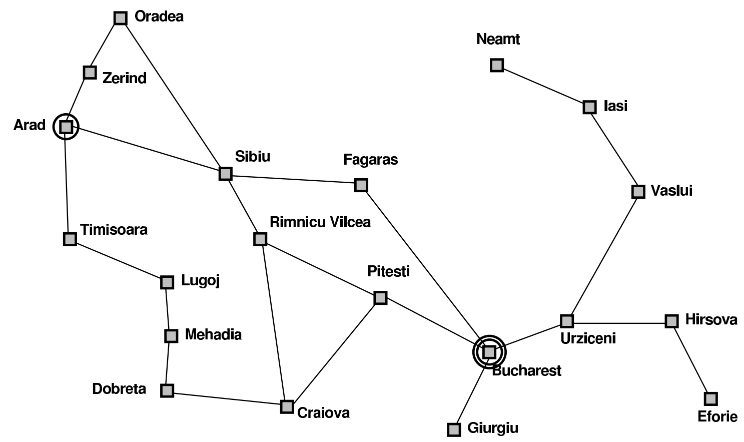
Single-state problem formulation:

- initial state
- operators or successor function
- goal test (explicit set of states or a predicate on states)
- path cost (additive)

Solution:

- A sequence of operators leading from initial state to a goal state

Example: Driving in Romania



Driving in Romania: Problem formulation

Problem formulation:

- initial state: Arad
- operators: {Arad → Zerind, Fagaras → Bucharest, Craiova → Pitesti, ...}
- goal test: the explicit set of states {Bucharest}
- path cost: total distance travelled so far

Solution:

- A sequence of operators: Arad → Sibiu → Fagaras → Bucharest

Selecting the right level of abstraction

- Abstract state (e.g., "in Arad") = set of real states
- Abstract operator (e.g., "Arad → Zerind") = complex combination of real actions
- Abstract solution (e.g., "Arad → Sibiu → Fagaras → Bucharest") = set of real-world paths/solutions

Abstraction should make the problem easier but the result should still be relevant

STATE-SPACE SEARCH

```
function SIMPLE-PROBLEM-SOLVING-AGENT(p) returns an action
inputs: p, a percept
static: s, an action sequence, initially empty
        state, some description of the current world state
        g, a goal, initially null
        problem, a problem formulation

state ← UPDATE-STATE(state, p)
if s is empty then
  g ← FORMULATE-GOAL(state)
  problem ← FORMULATE-PROBLEM(state, g)
  s ← SEARCH(problem)
action ← RECOMMENDATION(s, state)
s ← REMAINDER(s, state)
return action
```


SEARCH (CONT'D)

- Systematic, offline exploration of the state space
  - Expands states (i.e., generating successors of already-explored states)
  - according to some strategy

```
function GENERAL-SEARCH(problem, strategy) returns
  a solution or failure
initialize the search tree using the initial state of problem
loop
  if there are no candidates for expansion then return failure
  choose a leaf node for expansion according to strategy
  if the node contains a goal state then return corresponding solution
  else expand the node and add the resulting nodes to the search tree
forever
```


GENERAL IMPLEMENTATION

- Implement various strategies using a queue
  - In fact, various strategies are implemented by various variants of QUEUING-FN.
  - A strategy is defined by determining the order of node expansion

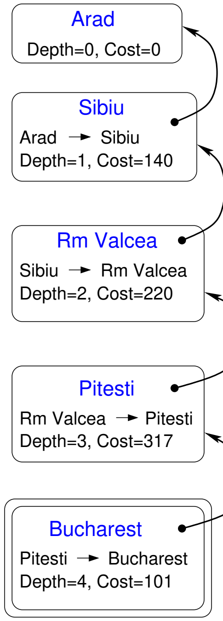
```
function GENERAL-SEARCH(problem, QUEUING-FN) returns
  a solution or failure

nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE(problem)))
loop
  if nodes is empty then return failure
  node ← REMOVE-FRONT(nodes)
  if GOAL-TEST(problem) applied to STATE(node) succeeds then
    return node
  nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS(problem)))
forever
```

- Other function of interest: MAKE-NODE (why?)

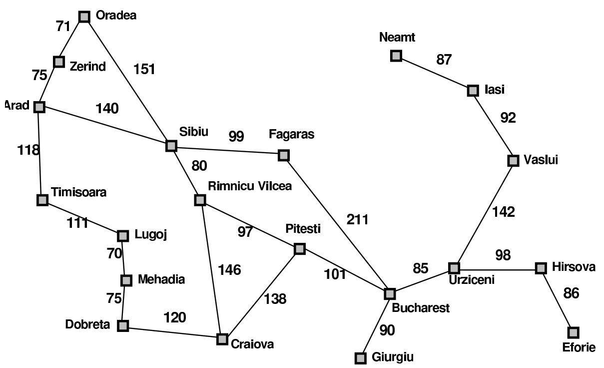
MAKE-NODE

A node definitely contains the state, but also:

- Its parent node
- The operator that was applied
- Its depth
- The path cost



Driving in Romania, revisited



Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Uninformed search

- Breadth-first:

function Queuing-Fn(*nodes, new-nodes*) **returns** queue of nodes
nodes ← Append(*nodes, new-nodes*)
end

- Depth-first;

function Queuing-Fn(*nodes, new-nodes*) **returns** queue of nodes
nodes ← Append(*new-nodes, nodes*)
end

- Uniform-cost:

function Queuing-Fn(*nodes, new-nodes*) **returns** queue of nodes
nodes ← Sort-By-Cost(Append(*nodes, new-nodes*))
end

- Depth-limited:
  - depth-first search with depth limit l
  - implementation: nodes at depth l have no children (successors)
- Iterative deepening:
  - Repeat depth-limited searches with depth l, for all $l > 0$, untill a good enough solution is found

Evaluation

Notations: branching factor: b; solution depth: d; maximum depth: m

	Complete?	Optimal?	Time	Space
breadth-first	Yes	Yes iff step cost=1	$O(b^d)$	$O(b^d)$
depth-first	No	No	$O(b^m)$	$O(bm)$
uniform cost	Yes iff step cost $\geq \varepsilon$	Yes	# of nodes with less than optimal path cost ($\simeq O(b^d)$)	# of nodes with less than optimal path cost ($\simeq O(b^d)$)
iterative deepening	Yes	Yes iff step cost=1	$O(b^d)$	$O(bd)$

Loop avoidance

- Operator
  - Do not generate parent
  - Follow the parent links and do not generate anything that is there already
- Search algorithm
  - Maintain a set of already expanded states
- Don't care

Informed search

Recall:

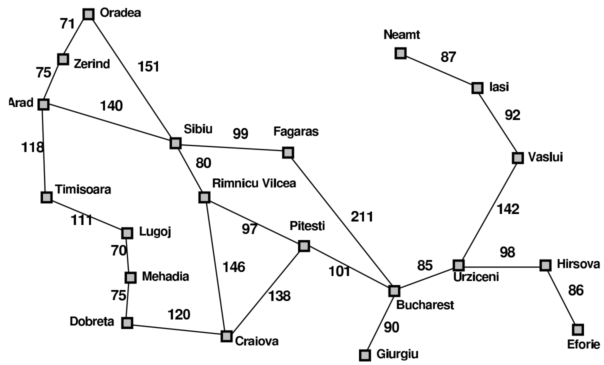
- General search algorithm.

```
function GENERAL-SEARCH(problem, QUEUING-FN) returns solution or
failure

nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE(problem)))
loop
   if nodes is empty then return failure
   node ← REMOVE-FRONT(nodes)
   if GOAL-TEST(problem) applied to STATE(node) succeeds then
      return node
   nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS(problem)))
forever
```

- Queueing discipline determines the order of expansion; in particular,

```
function QUEUING-FN(nodes, new-nodes) returns queue of nodes

nodes ← SORT-BY-PATH-COST(APPEND(nodes, new-nodes))
end
```


More driving



Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Best-first search

- Use an evaluation (heuristic) function for each node
- Always pick for expansion the most “desirable” node
- Implementation: Priority queue (insert nodes in decreasing order of desirability)
- Variants (of what?):
  - Greedy
  - A^*

Greedy search

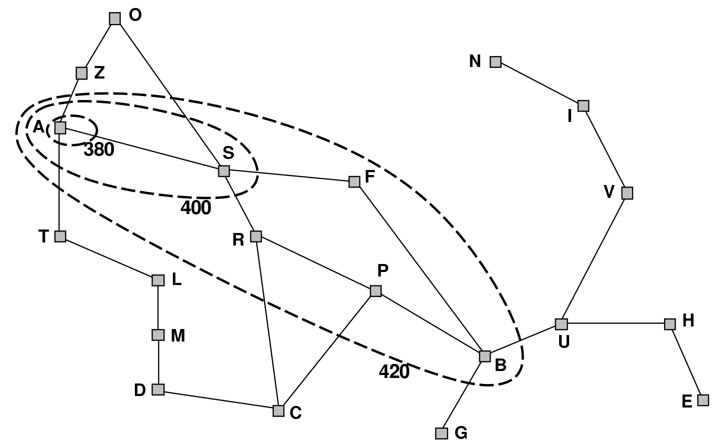
- Evaluation function $h(n)$ estimates cost from n to goal
  - E.g., $h_{sld}(n)$ = straight-line distance of n from Bucharest
- Greedy search expands first the node that appears to be closest to goal
  - Complete? No (can get stuck in loops)
    - Also prone to false starts
  - Optimal? No!
  - Time complexity? $O(b^m)$
  - Space Complexity? $O(b^m)$

A^* SEARCH

- Do not expand a path that is already expensive
- Two componends for the evaluation function: $f(n) = g(n) + h(n)$, where
  - $g(n)$ = const to reach n
  - $h(n)$ = estimated cost from n to goal
  - $f(n)$ = estimated cost from initial node to goal
- **Theorem.** If the heuristic h is admissible then A^* is optimal
  - A heuristic is admissible if it always underestimates the cost: $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost from n to goal

OPTIMALITY OF A^* (INFORMAL)

- Ideea: A^* expands nodes in order of increasing f values
  - Gradually adds "f-contours" of nodes (as breadth-first adds layers)



PROPERTIES

- Complete? Yes (for all practical purposes)
- Optimal? Yes
- Time complexity? Exponential in length of solution, error in h
- Space complexity? $O(b^d)$ (all nodes are kept in memory)

INVENTING HEURISTIC FUNCTIONS

- Admissible heuristics for the 8-puzzle:
  - h_1 the number of misplaced tiles
  - h_2 total Manhattan distance
- h_1 is always better than h_2, i.e., h_2 dominates h_1
  - thus A^* will expand fewer nodes when using h_2 than when using h_1
- Often, admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem
  - If the rules of the 8-puzzle are relaxed and a tile can move anywhere, then h_1 gives the shortest solution
  - If the rules of the 8-puzzle are relaxed and a tile can move to any adjacent square, then h_2 gives the shortest solution
  - Can you think of a good heuristic for TSP?

Inventing heuristic functions (cont'd)

- One can also invent heuristics using statistical information
  - The more information we gather in previous runs, the better the heuristic
  - However, we give up the guarantee of admisibility
- Need to pay attention to the computational complexity of the process of actually computing the heuristic function!

Other search methods

We will not cover these in the lectures, but you are supposed to take a look at the relevant material in the textbook

- Memory bounded variants of (mostly) A^* (Section 3.5.5)
- Iterative improvement algorithms (Sections 4.1)
  - Hill climbing
  - Simulated annealing
  - Local beam