

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.

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	<i>single-state problem</i>
deterministic, inaccessible	multiple-state problem
nondeterministic, inaccessible	contingency problem
unknown state space	exploration/online problem

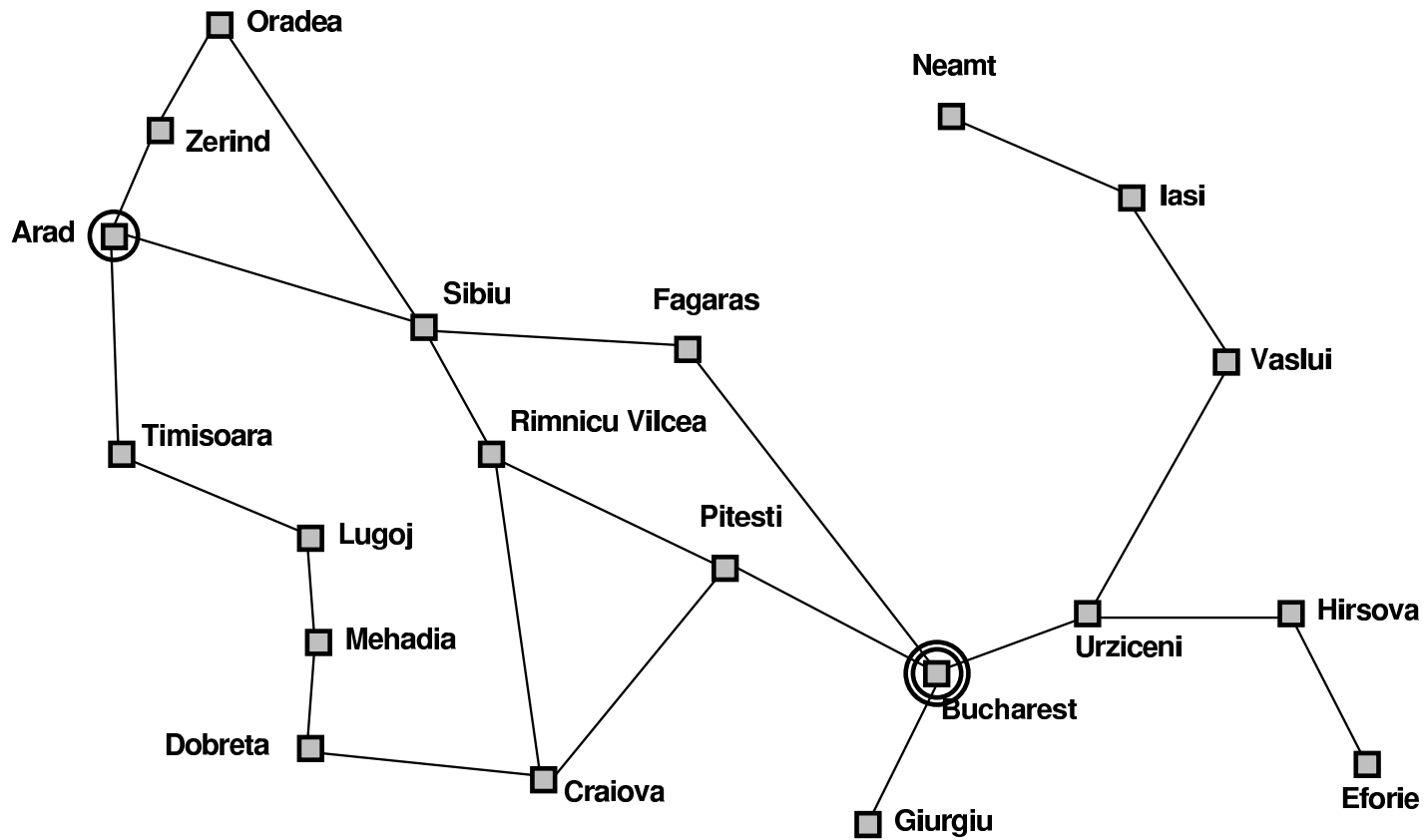
Single-state problem formulation:

- **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
 - ...by **expanding** 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 the 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**.

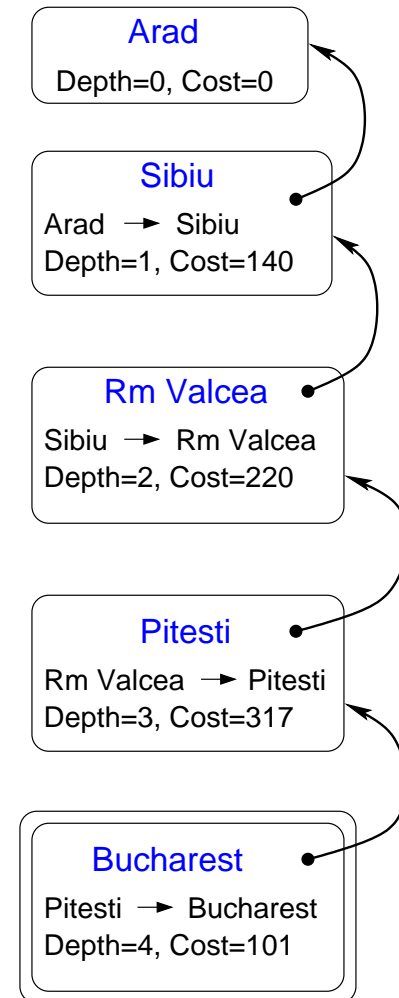
```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution, or failure
  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE(problem)))
  loop do
    if nodes is empty then return failure
    node ← REMOVE-FRONT(nodes)
    if GOAL-TEST(problem) applied to STATE(node) succeeds then return node
    nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS(problem)))
  end
```

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

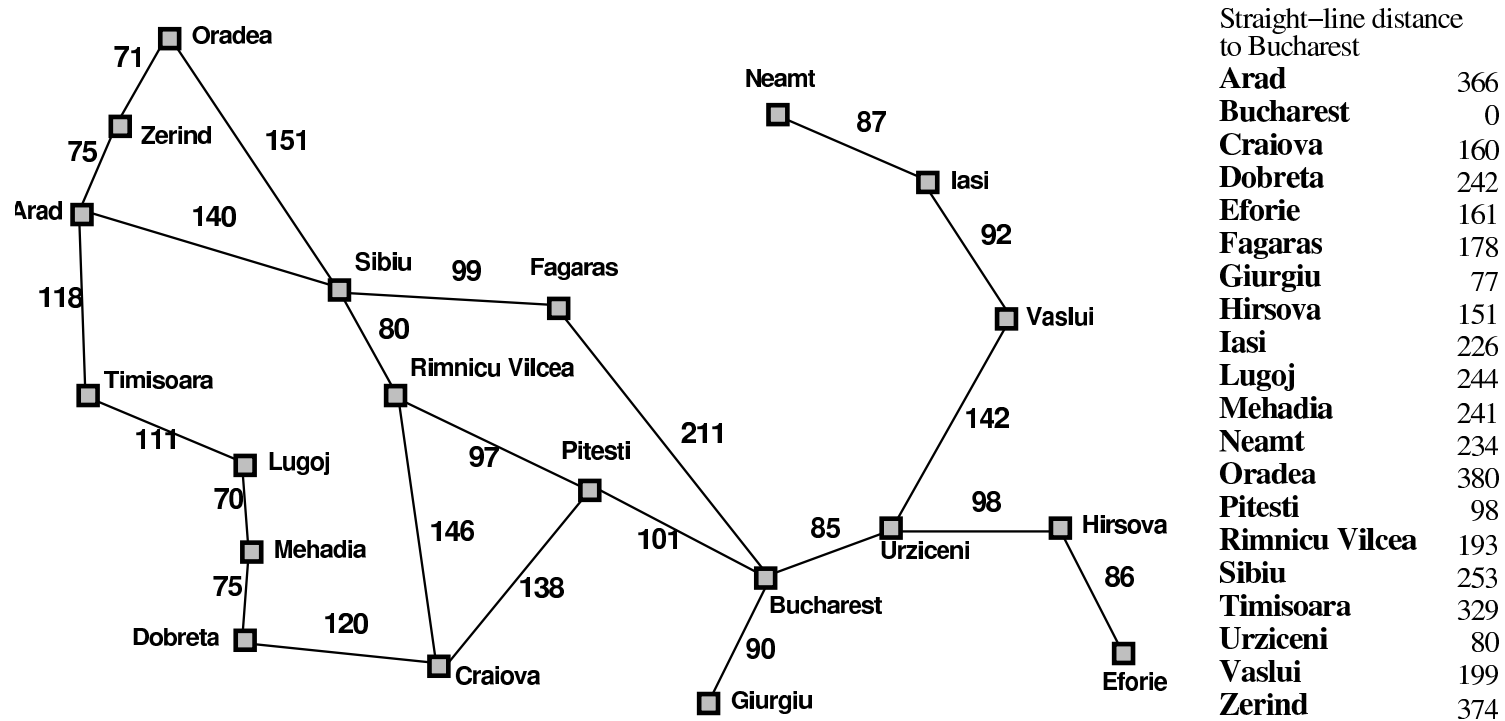
MAKE-NODE

A node contains the state alright, but also:

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



DRIVING IN ROMANIA, REPRISE



UNINFORMED SEARCH

- breadth-first

```
function QUEUING-FN( nodes, new-nodes) returns a new queue of nodes  
  nodes ← APPEND(nodes, new-nodes)  
end
```

UNINFORMED SEARCH

- breadth-first

```
function QUEUING-FN(nodes, new-nodes) returns a new queue of nodes  
  nodes ← APPEND(nodes, new-nodes)  
end
```

- depth-first

```
function QUEUING-FN(nodes, new-nodes) returns a new queue of nodes  
  nodes ← APPEND(new-nodes, nodes)  
end
```

UNINFORMED SEARCH

- breadth-first

```
function QUEUING-FN( nodes, new-nodes) returns a new queue of nodes  
  nodes ← APPEND(nodes, new-nodes)  
end
```

- depth-first

```
function QUEUING-FN( nodes, new-nodes) returns a new queue of nodes  
  nodes ← APPEND(new-nodes, nodes)  
end
```

- uniform-cost

```
function QUEUING-FN( nodes, new-nodes) returns a new queue of nodes  
  nodes ← SORT-BY-PATH-COST(APPEND(nodes, new-nodes))  
end
```

OTHER SEARCH METHODS

- **depth-limited search:**
 - depth-first search with depth limit l .
 - implementation: nodes at depth l have no children (successors).

OTHER SEARCH METHODS

- **depth-limited search:**
 - depth-first search with depth limit l .
 - implementation: nodes at depth l have no children (successors).
- **iterative deepening search:**
 - Repeatedly do depth-limited searches with depth l , for all $l > 0$, until a good enough solution is found.

PROPERTIES

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

	Complete?	Optimal?	Time	Space
breadth-first				
depth-first				
uniform cost				
iterative deepening				

PROPERTIES

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				
uniform cost				
iterative deepening				

PROPERTIES

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				
iterative deepening				

PROPERTIES

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 $\geq \epsilon$	Yes	no. of nodes with less than optimal path cost ($\simeq O(b^d)$)	no. of nodes with less than optimal path cost ($\simeq O(b^d)$)
iterative deepening				

PROPERTIES

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 $\geq \epsilon$	Yes	no. of nodes with less than optimal path cost ($\simeq O(b^d)$)	no. 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
- Don't care

INFORMED SEARCH

Recall:

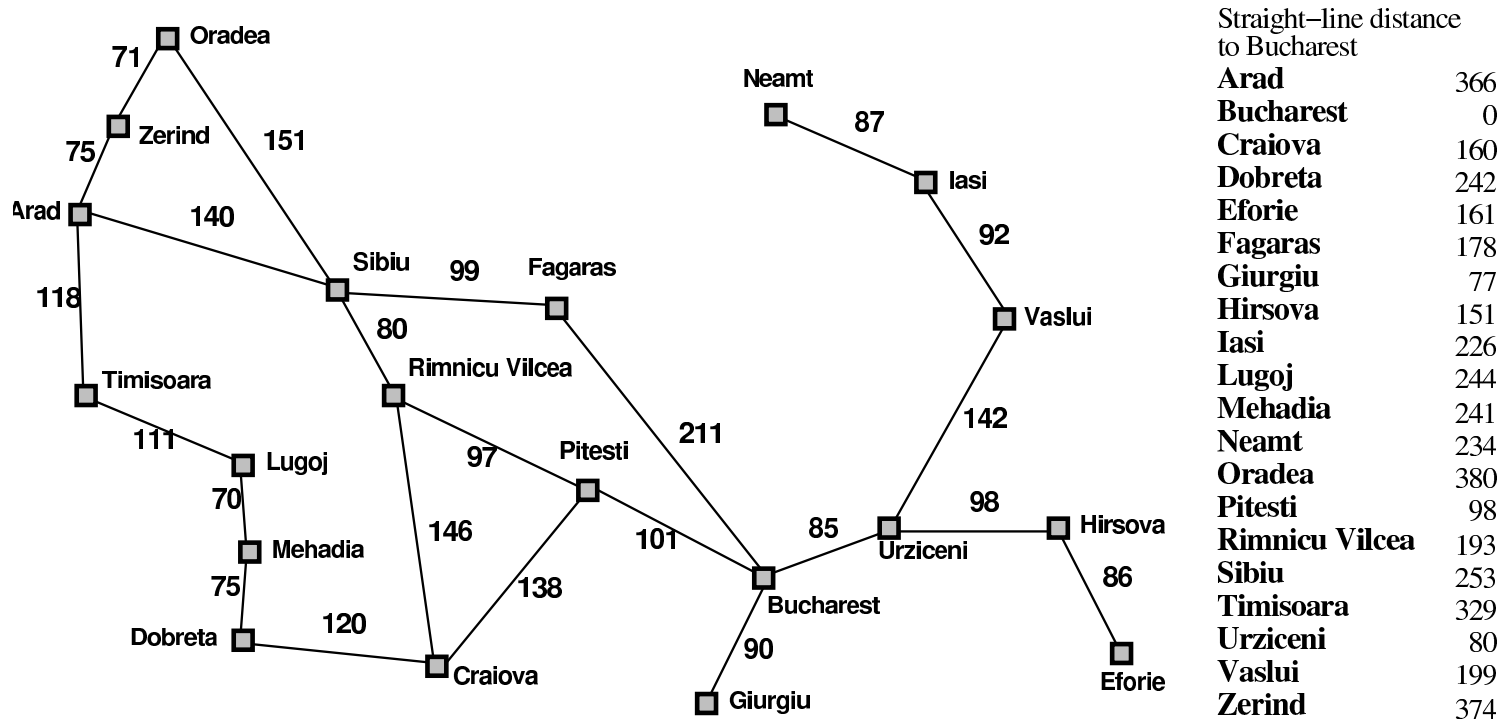
- General search algorithm.

```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution, or failure
  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE(problem)))
  loop do
    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)))
  end
```

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

```
function QUEUING-FN(nodes, new-nodes) returns a new queue of nodes
  nodes ← SORT-BY-PATH-COST(APPEND(nodes, new-nodes))
  end
```

THE MAP, AGAIN



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., $h_{slid}(n)$ = straight-line distance of n from Bucharest.
- Greedy search expands first the node that **appears** to be closest to goal.
 - Complete?
 - Optimal?
 - Time complexity?
 - Space Complexity?

GREEDY SEARCH

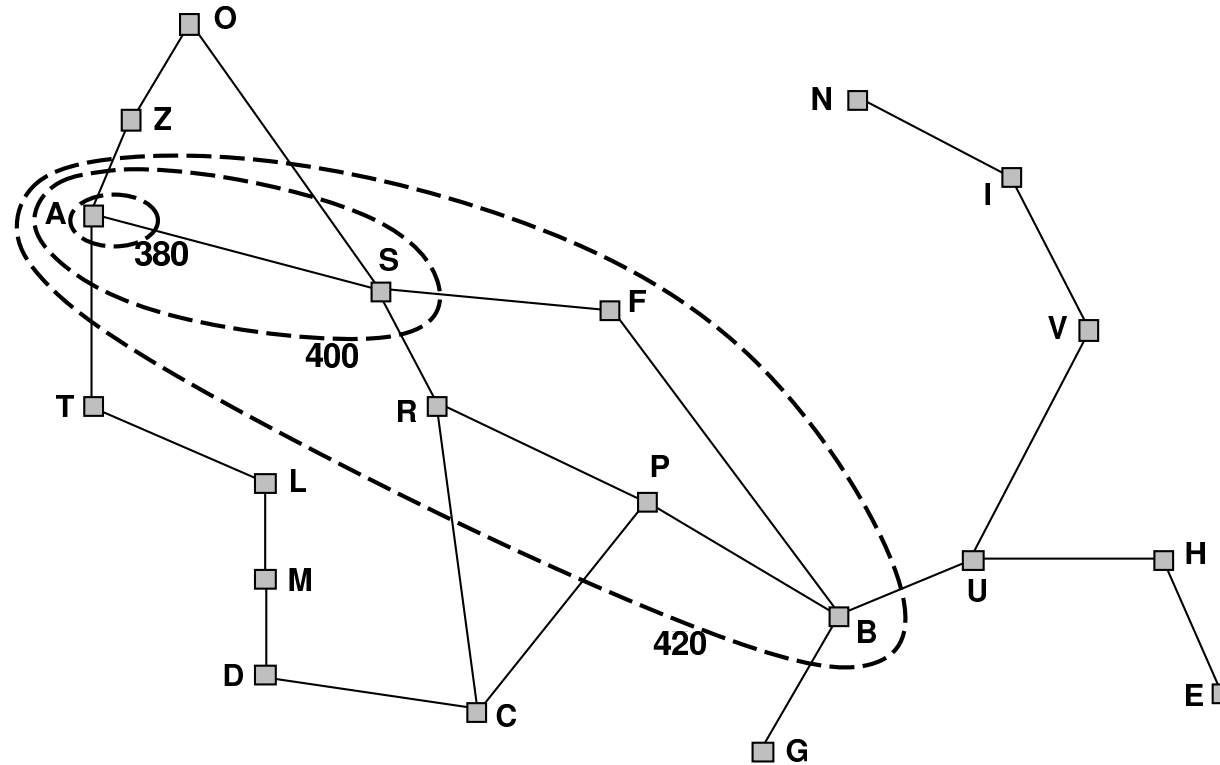
- Evaluation function $h(n)$ estimates cost from n to goal.
 - E.g., $h_{sld}(n)$ = straight-line distance of n from Bucharest.
- Greedy search expands first the node that **appears** to be closest to goal.
 - Complete? Can get stuck in loops.
 - * Also prone to false starts
 - Optimal? **No!**
 - Time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

A^* SEARCH

- Do not expand a path that is already expensive.
- Two components for the evaluation function: $f(n) = g(n) + h(n)$, where
 - $g(n) = \text{cost to reach } n$
 - $h(n) = \text{estimated cost from } n \text{ to goal.}$
 - $f(n) = \text{estimated cost from initial node to goal.}$
- **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)

- Idea: 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 (yay!)
- 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
 - but then we give up the guarantee of admissibility.
- 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.

- Iterative deepening A^* and SMA^* ;
(“Memory-bounded heuristic search” in Section 4.1 (2nd ed.) / Section 3.5.3 (3rd ed.))
- Iterative improvement algorithms
(Section 4.3 in 2nd ed. / Section 4.1 in 3rd ed.)
 - Hill-climbing
 - Simulated annealing