Artificial Intelligence

Problem Solving Agents

Outline

- Problem-solving agents
- Problem representation
- Problem formulation — state space
- Strategies for state space search

Problem Solving

- In order to cope, there are generally two ways:
    - Armor yourself and hope for the best (like a tree or a clam);
    - develop methods for getting out of harm's way and into the good's way.
- If taking the second method, then an agent must continually solve: Now what do I do? And usually a simple reflex agent won't be able to cope.
- We need a problem solving agent, which is a kind of goal-based agent. The goal is to solve a problem.

Another Definition of AI

- The study of representation and search through which intelligent activity can be enacted on a mechanical device.
- The function of a representation system: to capture the essential features of a problem domain and make that information accessible to a problem-solving agent.
  - Abstraction
  - Expressiveness
  - Efficiency

Representation Types

- Graph based
- Logic based
- Rule based
- Model based
- Case based
- Hybrid systems

Problem Solving Agent

- A problem solving agent usually is equipped with an internal representation system, and uses search strategies to solve a problem.
- For search algorithms the agent choose to use, we need to ask:
    - (completeness) Is the agent guaranteed to find a solution?
    - (termination) Will it always terminate, or can it be caught in an infinite loop?
    - (Optimality) Is its solution guaranteed to be optimal?
    - (Complexity) What is the cost (time and space complexity) of finding a solution?

Problem Solving Agent Types

- Offline ones --- find a solution and execute the solution with “eyes closed”.
- Online ones ---find the solution along with the execution. This becomes an exploration problem.

Problem types

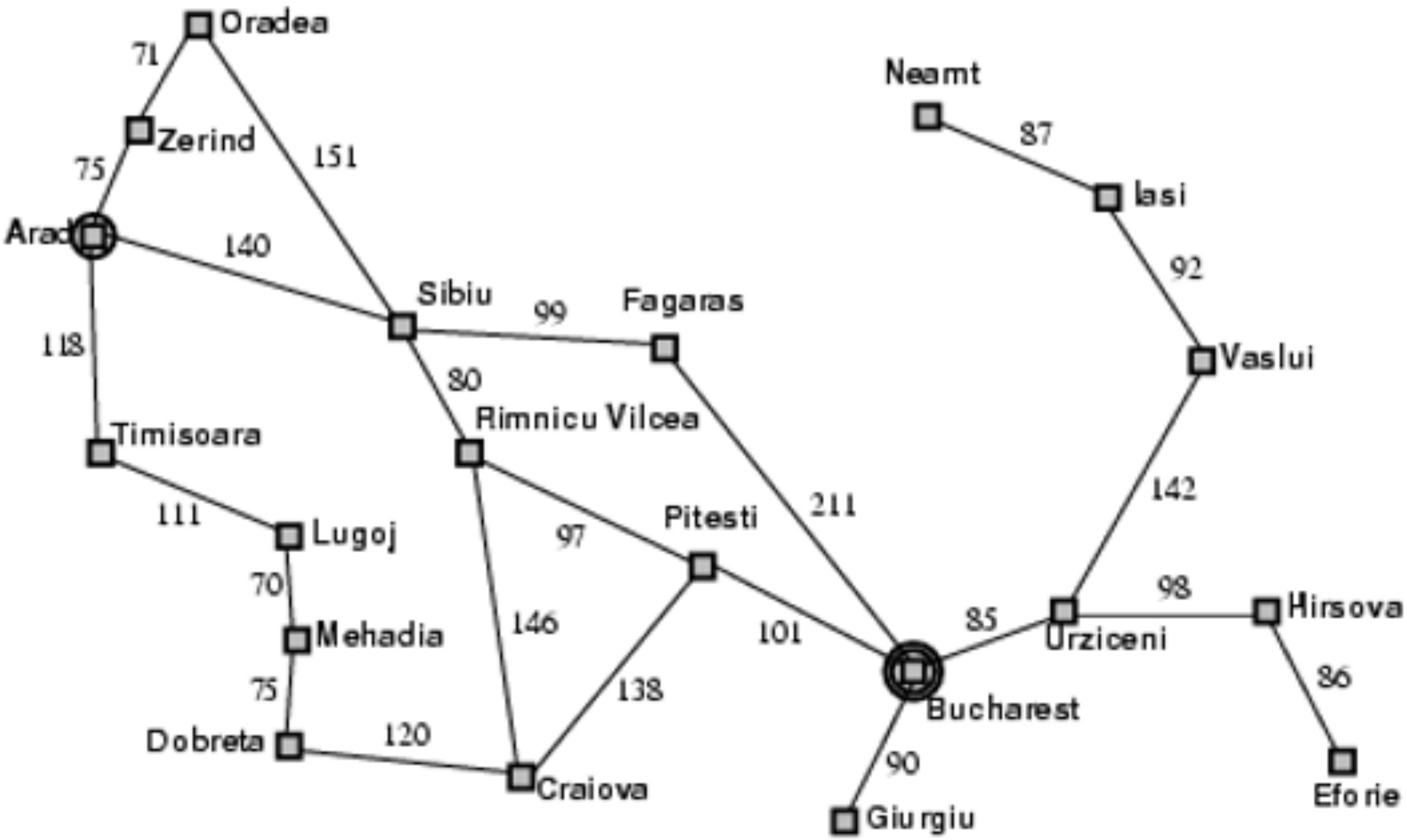
- Deterministic, fully observable —> single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence
- Non-observable —> sensorless problem (conformant problem)
  - Agent may have no idea where it is; solution is a sequence
- Nondeterministic and/or partially observable —> contingency problem
  - percepts provide new information about current state
  - often interleave search, execution
- Unknown state space —> exploration problem

Single State Problem's Representation system — State Space

A problem is defined by four items:

- initial state — where the agent starts in
- actions or successor function
  S(x) = set of <action, successor-state> pairs
  — where each action is one of the legal actions in state x and each successor state is a state that can be reached from x by applying the action
- goal test
  - Explicit (whether a given state is a goal state), e.g., x = Success
  - Implicit (whether a given goal is reached), e.g., Checkmate(x)
- path cost (additive) — the reflection of the performance measure

Example


Oradea
71
Neamt
Zerind
87
151
75
Iasi
Arad
140
92
Sibiu
99
Fagaras
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
211
Lugoj
97
Pitesti
70
98
85
Hirsova
Mehadia
146
101
Urziceni
86
75
138
Bucharest
Dobreta
120
90
Craiova
Eforie
Giurgiu

General Problem-solving agents

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

    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Selecting a state space

- Depends on
    - The specific problem, and
    - The internal representation of the agent
- Real world is absurdly complex — Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
- (Abstract) solution = set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem

State Space Graph

- State space is a graph based representation system.
- The initial state and the successor function together implicitly define the state space of the problem. It forms a graph.
  - Nodes — states
  - Arcs — actions (directed or undirected?)
  - Path — a sequence of states connected by a sequence of actions.
- A solution is a sequence of actions leading from the initial state to a goal state.
- Solving problem becomes systematically searching through state-space graph to find a path from initial state to goal state.
- Graph theory can be used to analyze the structure and complexity of both the problem and the search procedures used to solve it.

Strategies for State Space Search

- Two directions:
  - Data-Driven — From the given data (initial state) of a problem instance toward a goal
  - Goal-Driven — From a goal back to the data
- Types:
  - Uninformed search — search strategies use only the information available in the problem definition
  - Informed search — use heuristics

General Tree Search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
   successors ← the empty set
   for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
      s ← a new NODE
      PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
      PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
      DEPTH[s] ← DEPTH[node] + 1
      add s to successors
   return successors
```

Search strategies

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
    - completeness: does it always find a solution if one exists?
    - time complexity: number of nodes generated
    - space complexity: maximum number of nodes in memory
    - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
    - b: maximum branching factor of the search tree
    - d: depth of the least-cost solution
    - m: maximum depth of the state space (may be ∞)

Uninformed Search

- Search strategies use only the information available in the problem definition
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search

Summary of Uninformed Search Algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Informed Search Strategies

- Idea: use an evaluation function f(n) (usually involves heuristics) to estimate the “desirability” of candidate states.
- Implementation:
  - Order the candidate states in decreasing order of desirability
- Special cases:
  - greedy best-first search
  - A* search
- Heuristic refers to experience-based techniques for problem solving, learning, and discovery that gives a solution which is not guaranteed to be optimal.

Greedy best-first search

- Evaluation function $f(n) = h(n)$ (heuristic)
  $= $ estimate of cost from n to goal

- Greedy best-first search picks the state that appears to be closest to goal

Properties of greedy best-first search

- Complete?
  - No – can get stuck in loops, e.g., Iasi —> Neamt —> Iasi —> Neamt —> …
- Time?
  - $O(b^m)$, but a good heuristic can give dramatic improvement
- Space?
  - $O(b^m)$ -- keeps all nodes in memory
- Optimal?
  - No

A* search

- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal

Where do the heuristics come from? ---Relaxed problems

- A problem with fewer restrictions on the actions is called a relaxed problem

- The cost of an optimal solution to a relaxed problem is a heuristic for the original problem

- A heuristic $h(n)$ is admissible if for every node n, $h(n) \le h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.

- An admissible heuristic never over-estimates the cost to reach the goal, i.e., it is optimistic

- Theorem: If $h(n)$ is admissible, A* using TREE-SEARCH is optimal

Properties of A* Search

- Complete?
  - Yes (unless there are infinitely many nodes with $f \leq f(G)$)
- Time?
  - Exponential
- Space?
  - Keeps all nodes in memory
- Optimal?
  - Yes (if heuristics is admissible)

Local search algorithms

- Many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution, e.g., n-queens problem
- State space = set of "complete" configurations
- Find configuration satisfying constraints
- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it
- Algorithms:
    - Hill-climbing search -- "Like climbing Everest in thick fog with amnesia"
    - Simulated annealing search -- escape local maxima by allowing some "bad" moves but gradually decrease their frequency
    - Genetic algorithms

Genetic algorithms

- A successor state is generated by combining two parent states
- Start with k randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Use evaluation function (fitness function) -- higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation