Inf2D 02: Problem Solving by Searching

Stefano Albrecht

School of Informatics, University of Edinburgh

18/01/18

Slide Credits: Jacques Fleuriot, Michael Rovatsos, Michael Herrmann
Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms
Problem-solving agents

Agent has a “Formulate, Search, Execute” design

```plaintext
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    persistent: 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)
        if seq = failure then return a null action
        action ← FIRST(seq)
        seq ← REST(seq)
    return action
```
Example: Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest
- Formulate goal:
  - be in Bucharest
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest
Example: Romania
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
Example: vacuum world

- **Single-state**, start in #5.

Solution?
Example: vacuum world

- **Single-state**, start in \#5.

  Solution: \[ \text{Right, Suck} \]

- **Sensorless**, start in \{1, 2, 3, 4, 5, 6, 7, 8\} e.g., Right goes to \{2, 4, 6, 8\}

  Solution?
Example: vacuum world

- **Sensorless**, start in \{1, 2, 3, 4, 5, 6, 7, 8\} e.g., Right goes to \{2, 4, 6, 8\}

Solution: \[\text{[Right, Suck, Left, Suck]}\]

- **Contingency**
  - Nondeterministic: Suck may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: \([L, \text{Clean}]\), i.e., start in \#5 or \#7

Solution?
Example: vacuum world

- **Contingency**
  - Nondeterministic:
    Suck may dirty a clean carpet
  - Partially observable:
    location, dirt at current location.
  - Percept: \([\text{L, Clean}],\) i.e., start in \#5 or \#7

Solution: \([\text{Right, if dirt then Suck}]\)
A problem is defined by four items:

1. initial state e.g., “in Arad”
2. actions or successor function $S(x) =$ set of action–state pairs
   - e.g., $S(\text{Arad}) = \{\langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \ldots \}$
3. goal test, can be
   - explicit, e.g., $x =$ “in Bucharest"
   - implicit, e.g., $\text{Checkmate}(x)$
4. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - $c(x, a, y)$ is the step cost of taking action $a$ in state $x$ to reach state $y$, assumed to be $\geq 0$

- A solution is a sequence of actions leading from the initial state to a goal state
Selecting a state space

- Real world is absurdly complex → state space must be abstracted for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g., “Arad → Zerind” represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state “in Arad” must get to some real state "in Zerind"
- (Abstract) solution =
  - set of real paths that are solutions in the real world
- Each abstract action should be “easier” than the original problem
Vacuum world state space graph

- states?
- actions?
- goal test?
- path cost?
Vacuum world state space graph

- **states?** Pair of dirt and robot locations
- **actions?** Left, Right, Suck
- **goal test?** no dirt at any location
- **path cost?** 1 per action
Example: The 8-puzzle

- States?
- Actions?
- Goal test?
- Path cost?
Example: The 8-puzzle

- **states?** locations of tiles
- **actions?** move blank left, right, up, down
- **goal test?** = goal state (given)
- **path cost?** 1 per move
Example: robotic assembly

- **states?**: real-valued coordinates of robot joint angles & parts of the object to be assembled
- **actions?**: continuous motions of robot joints
- **goal test?**: complete assembly
- **path cost?**: time to execute
Tree search algorithms

Basic idea:

- offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. expanding states)

```
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 search example

[Diagram of a tree search example with nodes labeled Arad, Sibiu, Timisoara, Zerind, Fagaras, Oradea, Rimnicu Vlcea, Arad, Lugoj, Arad, Oradea]
Tree search example

Stefano Albrecht

Inf2D 02: Problem Solving by Searching
**Implementation: general tree search**

```plaintext
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
```

```plaintext
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)
```
A state is a (representation of) a physical configuration.

A node is a book-keeping data structure constituting part of a search tree. It includes state, parent node, action, path cost.

Using these it is easy to compute the components for a child node. (The CHILD-NODE function)
Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.