Multiagent Systems
Problem Solving and Uninformed Search

Viviana Mascardi
viviana.mascardi@unige.it

MAS, University of Genoa, DIBRIS
Disclaimer

- This presentation may contain material protected by copyright laws.
- In particular, many lessons are based on the material that Russel and Norvig made available on the Web, as part of their book “Artificial Intelligence: A Modern Approach”
- We made this material available on the Web only to ensure timely dissemination among the students of the INTS Course at the University of Genova, and is meant only for students’ personal use.
- Any use different from the students’ personal use is prohibited.
- M. Martelli and V. Mascardi thank Professor Schachte for the courtesy of sharing his teaching material on Prolog
1. Problem-solving agents

2. Problem types
   - Example: vacuum world

3. Problem formulation

4. Example problems
   - Vacuum world state space graph
   - The 8-puzzle

5. Basic search algorithms
   - Breadth-first search
   - Uniform-cost search
   - Depth-first search
   - Depth-limited search
   - Iterative deepening search
   - Summary of algorithms
Contents

1 Problem-solving agents

2 Problem types
   - Example: vacuum world

3 Problem formulation

4 Example problems
   - Vacuum world state space graph
   - The 8-puzzle

5 Basic search algorithms
   - Breadth-first search
   - Uniform-cost search
   - Depth-first search
   - Depth-limited search
   - Iterative deepening search
   - Summary of algorithms
Problem-solving agents

Restricted form of general agent:

```plaintext
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
        goal ← FORMULATE_GOAL(state)
        problem ← FORMULATE_PROBLEM(state, goal)
        seq ← SEARCH(problem)
        action ← RECCOMENDATION(seq,state)
        seq ← REMAINDER(seq,state)
    return action
```

This is offline problem solving; solution executed “eyes closed”. 
Online problem solving involves acting without complete knowledge.
Problem-solving agents

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
Contents

1. Problem-solving agents
2. Problem types
   - Example: vacuum world
3. Problem formulation
4. Example problems
   - Vacuum world state space graph
   - The 8-puzzle
5. Basic search algorithms
   - Breadth-first search
   - Uniform-cost search
   - Depth-first search
   - Depth-limited search
   - Iterative deepening search
   - Summary of algorithms
Problem types

Deterministic, fully observable $\implies$ single-state problem
Agent knows exactly which state it will be in; solution is a sequence

Non-observable $\implies$ conformant problem
Agent may have no idea where it is; solution (if any) is a sequence

Nondeterministic $\implies$ contingency problem
percepts provide new information about current state
solution is a contingent plan or a policy often interleave search, execution

Unknown state space $\implies$ exploration problem ("online")
Example: vacuum world

Single-state: start in #5
Solution??
Example: vacuum world

Single-state: start in #5
Solution??

[Right, Suck]
Example: vacuum world

Single-state: start in #5
Solution??
[Right, Suck]

Conformant: start in \( \{1, 2, 3, 4, 5, 6, 7, 8\} \)
e.g., Right goes to \( \{2, 4, 6, 8\} \)
Solution??
Example: vacuum world

Single-state: start in #5
Solution??
[Right, Suck]

Conformant: start in \{1, 2, 3, 4, 5, 6, 7, 8\}
e.g., Right goes to \{2, 4, 6, 8\}
Solution??
[Right, Suck, Left, Suck]
Example: vacuum world

**Single-state:** start in #5
Solution??
\[ \text{[Right, Suck]} \]

**Conformant:** 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:** start in #5
Murphy’s Law: Suck can dirty a clean carpet
Local Sensing: dirt, location only
Solution??
Example: vacuum world

Single-state: start in \#5
Solution??
[Right, Suck]

Conformant: start in \{1, 2, 3, 4, 5, 6, 7, 8\}
e.g., Right goes to \{2, 4, 6, 8\}
Solution??
[Right, Suck, Left, Suck]

Contingency: start in \#5
Murphy’s Law: Suck can dirty a clean carpet
Local Sensing: dirt, location only
Solution??
[Right, if dirt then Suck]
Contents

1 Problem-solving agents

2 Problem types
   - Example: vacuum world

3 Problem formulation

4 Example problems
   - Vacuum world state space graph
   - The 8-puzzle

5 Basic search algorithms
   - Breadth-first search
   - Uniform-cost search
   - Depth-first search
   - Depth-limited search
   - Iterative deepening search
   - Summary of algorithms
Single-state problem formulations

A problem is defined by four items:

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

A solution is a sequence of actions leading from the initial state to a goal state.
Problem formulation

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!
Contents

1 Problem-solving agents
2 Problem types
   • Example: vacuum world
3 Problem formulation
4 Example problems
   • Vacuum world state space graph
   • The 8-puzzle
5 Basic search algorithms
   • Breadth-first search
   • Uniform-cost search
   • Depth-first search
   • Depth-limited search
   • Iterative deepening search
   • Summary of algorithms
Example: vacuum world state space graph
Example: vacuum world state space graph

States??:
Example problems

Example: vacuum world state space graph

States??: integer dirt and robot location (ignore dirt amounts etc.)
Example: vacuum world state space graph

States??: integer dirt and robot location (ignore dirt amounts etc.)

Actions??:
Example: vacuum world state space graph

States??: integer dirt and robot location (ignore dirt amounts etc.)
Actions??: Left, Right, Suck, NoOp
Example: vacuum world state space graph

States???: integer dirt and robot location (ignore dirt amounts etc.)
Actions???: Left, Right, Suck, NoOp
Goal Test???:

MAS, University of Genoa, DIBRIS
Example problems

Example: vacuum world state space graph

States???: integer dirt and robot location (ignore dirt amounts etc.)
Actions???: *Left*, *Right*, *Suck*, *NoOp*
Goal Test???: no dirt
Example: vacuum world state space graph

States???: integer dirt and robot location (ignore dirt amounts etc.)
Actions???: *Left*, *Right*, *Suck*, *NoOp*
Goal Test???: no dirt
Path Cost???:

---

MAS, University of Genoa, DIBRIS
Example: vacuum world state space graph

States??: integer dirt and robot location (ignore dirt amounts etc.)
Actions??: *Left*, *Right*, *Suck*, *NoOp*
Goal Test??: no dirt
Path Cost??: 1 per action (0 for *NoOp*)
Example: The 8-puzzle

Start State

Goal State
Example: The 8-puzzle

Start State

Goal State

States??:
Example problems

Example: The 8-puzzle

States??: integer locations of tiles (ignore intermediate positions.)
Example: The 8-puzzle

States??: integer locations of tiles (ignore intermediate positions.)
Actions??:
Example: The 8-puzzle

States???: integer locations of tiles (ignore intermediate positions.)
Actions???: move blank left, right, up, down (ignore unjamming etc.)
Example: The 8-puzzle

States??: integer locations of tiles (ignore intermediate positions.)
Actions??: move blank left, right, up, down (ignore unjamming etc.)
Goal Test??:
Example: The 8-puzzle

States??: integer locations of tiles (ignore intermediate positions.)
Actions??: move blank left, right, up, down (ignore unjamming etc.)
Goal Test??: goal state (given)
Example: The 8-puzzle

States??: integer locations of tiles (ignore intermediate positions.)
Actions??: move blank left, right, up, down (ignore unjamming etc.)
Goal Test??: goal state (given)
Path Cost??:
Example: The 8-puzzle

States??: integer locations of tiles (ignore intermediate positions.)
Actions??: move blank left, right, up, down (ignore unjamming etc.)
Goal Test??: goal state (given)
Path Cost??: 1 per move
Example: The 8-puzzle

States???: integer locations of tiles (ignore intermediate positions.)
Actions???: move blank left, right, up, down (ignore unjamming etc.)
Goal Test???: goal state (given)
Path Cost???: 1 per move
Note: optimal solution of $n$-puzzle is NP-hard
Contents

1. Problem-solving agents
2. Problem types
   - Example: vacuum world
3. Problem formulation
4. Example problems
   - Vacuum world state space graph
   - The 8-puzzle
5. Basic search algorithms
   - Breadth-first search
   - Uniform-cost search
   - Depth-first search
   - Depth-limited search
   - Iterative deepening search
   - Summary of algorithms
Basic search algorithms

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

```plaintext
function TREE-SEARCH(problem, strategy) returns a solution, or failure
initialize the search tree using the initial state of problem
loop do
    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
end
```
Problem types
Basic search algorithms

Problem types

Diagram showing a tree structure with cities such as Arad, Sibiu, Timisoara, and Zerind connected by lines.
Problem types
A state is a (representation of) a physical configuration
A node is a data structure constituting part of a search tree includes
parent, children, depth, path cost \( g(x) \)
States do not have parents, children, depth, or path cost!

The Expand function creates new nodes, filling in the various fields and using the Successor of the problem to create the corresponding states.
function TREE-SEARCH(problem, strategy) 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 node
  fringe ← INSERTALL(EXPAND(node, problem), fringe)
end

function EXPAND(node, problem) returns a set of nodes

successors ← the empty set

foreach action, result in SUCCESSOR-FN(problem, STATE[node]) do
  s ← 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
A 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/expanded
- **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 $\infty$)
Uninformed search strategies

Uninformed 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
Breadth-first search

Expand shallowest unexpanded node

Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end
Breadth-first search

Expand shallowest unexpanded node

Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end
Breadth-first search

Expand shallowest unexpanded node

Implementation:
*fringe* is a FIFO queue, i.e., new successors go at end
Breadth-first search

Expand shallowest unexpanded node

Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end
Properties of breadth-first search

Complete??
Properties of breadth-first search

**Complete** - Yes (if $b$ is finite)
Properties of breadth-first search

Complete - Yes (if $b$ is finite)
Time??
Properties of breadth-first search

Complete - Yes (if $b$ is finite)
Time - $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in $d$ Space??
Properties of breadth-first search

Complete - Yes (if $b$ is finite)
Time - $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = \mathcal{O}(b^{d+1})$, i.e., exp. in $d$
Space - $\mathcal{O}(b^{d+1})$ (keeps every node in memory)
Properties of breadth-first search

Complete - Yes (if $b$ is finite)
Time - $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in $d$
Space - $O(b^{d+1})$ (keeps every node in memory)
Optimal??
Properties of breadth-first search

Complete - Yes (if $b$ is finite)
Time - $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in $d$
Space - $O(b^{d+1})$ (keeps every node in memory)
Optimal - Yes (if cost = 1 per step); not optimal in general
Properties of breadth-first search

Complete - Yes (if $b$ is finite)
Time - $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in $d$
Space - $O(b^{d+1})$ (keeps every node in memory)
Optimal - Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8640GB.
Uniform-cost search

Expand least-cost unexpanded node
Implementation: \textit{fringe} = queue ordered by path cost, lowest first
Equivalent to breadth-first if step costs all equal

Complete Yes, if step cost $\geq \epsilon$
Time # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$ where $C^*$ is the cost of the optimal solution
Space # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
Optimal Yes—nodes expanded in increasing order of $g(n)$
Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:

\[ \text{fringe} = \text{LIFO queue, i.e., put successors at front} \]
Depth-first search

Expand deepest unexpanded node

Implementation:
fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:

*fringe* = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:

\textit{fringe} = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:

*fringe* = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:

*fringe* = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:

*fringe* = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation: 
fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:

\textit{fringe} = LIFO queue, i.e., put successors at front
Properties of depth-first search

Complete??
Properties of depth-first search

Complete - No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
⇒ complete in finite spaces
Properties of depth-first search

**Complete** - No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
⇒ complete in finite spaces

Time??
Properties of depth-first search

Complete - No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
⇒ complete in finite spaces

Time - $O(b^m)$: terrible if $m$ is much larger than $d$ but if solutions are dense, may be much faster than breadth-first

Space??
Properties of depth-first search

Complete - No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
⇒ complete in finite spaces

Time - $O(b^m)$: terrible if $m$ is much larger than $d$ but if solutions are dense, may be much faster than breadth-first

Space - $O(bm)$, i.e., linear space!
Properties of depth-first search

**Complete** - No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
⇒ complete in finite spaces

**Time** - $O(b^m)$: terrible if $m$ is much larger than $d$ but if solutions are dense, may be much faster than breadth-first

**Space** - $O(bm)$, i.e., linear space!

Optimal??
Properties of depth-first search

**Complete** - No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
⇒ complete in finite spaces

**Time** - $O(b^m)$: terrible if $m$ is much larger than $d$ but if solutions are dense, may be much faster than breadth-first

**Space** - $O(bm)$, i.e., linear space!

**Optimal** - No
Depth-limited search

= depth-first search with depth limit \( l \),
i.e., nodes at depth \( l \) have no successors

Recursive implementation:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
cutoff-occurred? ← false
if GOAL-TEST(problem, STATE[node]) then return node
else if DEPTH[node] = limit then return cutoff
else foreach successor in EXPAND(node, problem) do
    result ← RECURSIVE-DLS(successor, problem, limit) 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
inputs problem, a problem

for depth ← 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
Iterative deepening search

Limit = 0
Iterative deepening search

Limit = 1

Diagram showing the iterative deepening search process.
Iterative deepening search

Limit = 2

[Diagram of iterative deepening search process]
Iterative deepening search

Limit = 3
Properties of iterative deepening search

Complete??
Properties of iterative deepening search

Complete - Yes
Properties of iterative deepening search

Complete - Yes
Time??
Properties of iterative deepening search

Complete - Yes
Time - $(d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)$
Space??
Properties of iterative deepening search

Complete - Yes

Time - \((d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)\)

Space - \(O(bd)\)
Properties of iterative deepening search

Complete - Yes
Time - \((d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)\)
Space - \(O(bd)\)
Optimal??
Properties of iterative deepening search

Complete - Yes
Time - \((d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)\)
Space - \(O(bd)\)
Optimal - Yes, if step cost = 1.
Can be modified to explore uniform-cost tree
Properties of iterative deepening search

Complete - Yes
Time - \((d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)\)
Space - \(O(bd)\)
Optimal - Yes, if step cost = 1.
Can be modified to explore uniform-cost tree

Numerical comparison for \(b = 10\) and \(d = 5\), solution at far right leaf:

\[N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450\]
\[N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100\]

IDS does better because other nodes at depth \(d\) are not expanded
BFS can be modified to apply goal test when a node is generated
## Summary of algorithms

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breath-First</th>
<th>Uniform-Cost</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>Yes*</td>
<td>Yes*</td>
<td>No</td>
<td>Yes, if ( l \geq d )</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>( b^{d+1} )</td>
<td>( b^{\lceil C^*/\epsilon \rceil} )</td>
<td>( b^m )</td>
<td>( b^l )</td>
<td>( b^d )</td>
</tr>
<tr>
<td>Space</td>
<td>( b^{d+1} )</td>
<td>( b^{\lceil C^*/\epsilon \rceil} )</td>
<td>( b^m )</td>
<td>( b^l )</td>
<td>( bd )</td>
</tr>
<tr>
<td>Optimal</td>
<td>Yes*</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes*</td>
</tr>
</tbody>
</table>
Repeated states

Failure to detect repeated states can turn a linear problem into an exponential one!
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
  closed ← an empty set
  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 node
  if STATE(node) is not closed then
    add STATE[node] to closed
    fringe ← INSERTALL(EXPAND(node, problem), fringe)
end
Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.

- Variety of uninformed search strategies.

- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms.

- Graph search can be exponentially more efficient than tree search.