Chap 3: Problem-solving and search
Outline

- Problem-solving agents
  - A kind of goal-based agent

- Problem types
  - Single state (fully observable)
  - Search with partial information

- Problem formulation
  - Example problems

- Basic search algorithms
  - Uninformed
Problem-solving agent

- Four general steps in problem solving:
  - **Goal formulation**
    - What are the successful world states
  - **Problem formulation**
    - What actions and states to consider given the goal
  - **Search**
    - Determine the possible sequence of actions that lead to the states of known values and then choosing the best sequence.
  - **Execute**
    - Given the solution perform the actions.
Problem-solving agent

function SIMPLE-PROBLEM-SOLVING-AGENT(percept) return an action

static: seq, an action sequence
    state, some description of the current world state
    goal, a goal
    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 ← FIRST(seq)
    seq ← REST(seq)
return action
Example: Romania
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, ...
Problem formulation

A problem is defined by:

- **An initial state**, e.g. *Arad*
- **Successor function** $S(X) = \text{set of action-state pairs}$
  - e.g. $S(Arad)=\{<\text{Arad} \rightarrow \text{Zerind}, \text{Zerind}>, \ldots\}$
- **Goal test**, can be
  - Explicit, e.g. $x='\text{at bucharest}'$
  - Implicit, e.g. $\text{checkmate}(x)$
- **Path cost** *(additive)*
  - e.g. sum of distances, number of actions executed, …
  - $c(x,a,y)$ is the step cost of going from state $x$ to state $y$ by performing action $a$, $c(x,a,y)$ is assumed to be $\geq 0$

*A solution* is a sequence of actions from initial to goal state.

**Optimal solution** has the lowest path cost.
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.
  - The abstraction is valid if the path between two states is reflected in the real world.

- (Abstract) solution <= set of real paths that are solutions in the real world.

- Each abstract action should be “easier” than the real problem.
Example: vacuum world

- States??
- Initial state??
- Actions??
- Goal test??
- Path cost??
Example: vacuum world

- States?? two locations with or without dirt: $2 \times 2^2 = 8$ states.
- Initial state?? Any state can be initial
- Actions?? \{Left, Right, Suck\}
- Goal test?? Check whether both squares are clean.
- Path cost?? Number of actions to reach goal. (Assume step costs have the same weight for the three actions.)
Example: 8-puzzle

- States??
- Initial state??
- Actions??
- Goal test??
- Path cost??

![Start State](image1)
![Goal State](image2)
Example: 8-puzzle

- States?? Location of the eight tiles (board configuration)
- Initial state?? Any state can be initial
- Actions?? \{Left, Right, Up, Down\} movements of the blank position
- Goal test?? Check whether goal configuration is reached
- Path cost?? Number of actions to reach goal
Example: 8-queens problem

- States??
- Initial state??
- Actions??
- Goal test??
- Path cost??
Example: 8-queens problem

Incremental formulation vs. complete-state formulation

- States??
- Initial state??
- Actions??
- Goal test??
- Path cost??
Example: 8-queens problem

Incremental formulation

- States?? Any arrangement of 0 to 8 queens on the board
- Initial state?? No queens
- Actions?? Add one queen in any empty square
- Goal test?? 8 queens on board and none attacked
- Path cost?? None (for this problem, the path cost is irrelevant.)

$3 \times 10^{14}$ possible sequences to investigate
Example: 8-queens problem

Incremental formulation (alternative)

- States?? \( n \) (\( 0 \leq n \leq 8 \)) queens on the board, one per column in the \( n \) leftmost columns with no queen attacking another.

- Actions?? Add one queen in leftmost empty column such that it is not attacking other queens

- Possible board configurations: 8x8x8x8x8x8x8x8 (not every one is legal state.)

- State space: 2,057 legal states
Example: robot assembly

- States??
- Initial state??
- Actions??
- Goal test??
- Path cost??
Example: robot assembly

- States?? Real-valued coordinates of robot joint angles; parts of the object to be assembled.
- Initial state?? Any arm position and object configuration.
- Actions?? Continuous motion of robot joints
- Goal test?? Complete assembly (don’t care robot joint angles)
- Path cost?? Time to execute
Basic search algorithms

How do we find the solutions of previous problems?

- **Search the state space**
  - complexity of space depends on state representation

- **Here: search through** explicit tree generation
  - ROOT= initial state.
  - Nodes and leafs generated through successor function.

- **In general search generates a graph** (same state through multiple paths)
function TREE-SEARCH(problem, strategy) return a solution or failure

    Initialize search tree to the initial state of the problem
    do
        if no candidates for expansion then return failure
        choose leaf node for expansion according to strategy
        if node contains goal state then return solution
        else expand the node and add resulting nodes to the search tree
    enddo
Simple tree search example

function TREE-SEARCH(problem, strategy) return a solution or failure

Initialize search tree to the initial state of the problem

do

if no candidates for expansion then return failure
choose leaf node for expansion according to strategy
if node contains goal state then return solution
else expand the node and add resulting nodes to the search tree

endo
Simple tree search example

function TREE-SEARCH(problem, strategy) return a solution or failure

Initialize search tree to the initial state of the problem

do

if no candidates for expansion then return failure
choose leaf node for expansion according to strategy
if node contains goal state then return solution
else expand the node and add resulting nodes to the search tree
enddo

← Determines search process!!
State space vs. search tree

- A state is a (representation of) physical configuration
- A node is a data structure belong to a search tree
  - A node has a parent, children, ... and includes path cost, depth, ...
  - Here node = <state, parent-node, action, path-cost, depth>
  - FRINGE = contains generated nodes which are not yet expanded.
    - White nodes with black outline in the search tree
Tree search algorithm

function TREE-SEARCH(problem, fringe) return a solution or failure

fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)

loop do

if EMPTY?(fringe) then return failure

node ← REMOVE-FIRST(fringe)

if GOAL-TEST[problem] applied to STATE[node] succeeds

then return SOLUTION(node)

/* SOLUTION function returns the sequence of actions obtained
by following parent pointers back to the root.*/

fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
Tree search algorithm (2)

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

    successors ← the empty set

    for each <action, result> in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        STATE[s] ← result
        PARENT-NODE[s] ← node
        ACTION[s] ← action
        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 strategy is defined by picking the order of node expansion.
- Problem-solving performance is measured in four ways:
  - **Completeness;** Does it always find a solution if one exists?
  - **Optimality;** Does it always find the least-cost solution?
  - **Time Complexity;** Number of nodes generated/expanded?
  - **Space Complexity;** Number of nodes stored in memory during search?

- Time and space complexity are measured in terms of problem difficulty defined by:
  - $b$ - branching factor or maximum # of successors of any node of the search tree
  - $d$ - depth of the least-cost solution (optimal solution)
  - $m$ - maximum depth of the state space (may be $\infty$)
Uninformed search strategies

- (a.k.a. blind search) = use only information available in problem definition.
  - When strategies can determine whether one non-goal state is better than another → informed search.

- Categories defined by expansion algorithm:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search.
  - Bidirectional search
BF-search, an example

- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue
BF-search, an example

- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue
BF-search, an example

- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue

```
  A
 / \       /
B   C
|   |      |   |
D   E      C
```


BF-search, an example

- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue
BF-search; evaluation

Completeness:

- *Does it always find a solution if one exists?*
- **YES**
  - If shallowest goal node is at some finite depth $d$
  - Condition: If $b$ is finite
    - *(maximum num. of successor nodes is finite)*
BF-search; evaluation

- Completeness:
  - YES (if $b$ is finite)

- Time complexity (worst case):
  - Assume a state space where every state has $b$ successors.
    - root has $b$ successors, each node at the next level has again $b$ successors (total $b^2$), …
    - Assume solution is at depth $d$
    - Worst case; expand all but the last node (contains goal) at depth $d$
    - Total num. of nodes generated:
      $$b + b^2 + b^3 + ... + b^d + (b^{d+1} - b) = O(b^{d+1})$$
BF-search; evaluation

- Completeness:
  - YES (if $b$ is finite)

- Time complexity:
  - Total numb. of nodes generated:
    \[ b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1}) \]

- Space complexity:
  - Same as time complexity if each node is retained in memory
BF-search; evaluation

- Completeness:
  - YES (if $b$ is finite)

- Time complexity:
  - Total numb. of nodes generated:
    $$b + b^2 + b^3 + ... + b^d + (b^{d+1} - b) = O(b^{d+1})$$

- Space complexity:
  - Same as time complexity if each node is retained in memory

- Optimality:
  - Does it always find the least-cost solution?
    - YES if step cost is the same for all actions
    - NO if actions have different cost.
BF-search; evaluation

Two lessons:

- Memory requirements are a bigger problem than its execution time.
- Exponential complexity search problems cannot be solved by uninformed search methods for any but the smallest instances.

<table>
<thead>
<tr>
<th>DEPTH</th>
<th>NODES</th>
<th>TIME</th>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1100</td>
<td>0.11 seconds</td>
<td>1 megabyte</td>
</tr>
<tr>
<td>4</td>
<td>111100</td>
<td>11 seconds</td>
<td>106 megabytes</td>
</tr>
<tr>
<td>6</td>
<td>$10^7$</td>
<td>19 minutes</td>
<td>10 gigabytes</td>
</tr>
<tr>
<td>8</td>
<td>$10^9$</td>
<td>31 hours</td>
<td>1 terabyte</td>
</tr>
<tr>
<td>10</td>
<td>$10^{11}$</td>
<td>129 days</td>
<td>101 terabytes</td>
</tr>
<tr>
<td>12</td>
<td>$10^{13}$</td>
<td>35 years</td>
<td>10 petabytes</td>
</tr>
<tr>
<td>14</td>
<td>$10^{15}$</td>
<td>3523 years</td>
<td>1 exabyte</td>
</tr>
</tbody>
</table>

$b = 10; \ 10,000 \ \text{nodes/sec; 1,000 \ \text{bytes/node}}$
Uniform-cost search

- Extension of BF-search:
  - Expand node with \textit{lowest path cost}

- Implementation: \textit{fringe} = queue ordered by path cost.

- UC-search is the same as BF-search when all step-costs are equal.
Uniform-cost search

- Completeness:
  - **YES**, if step-cost > ε (some small positive constant)

- Time complexity:
  - Assume C* the cost of the optimal solution.
  - Assume that every action costs at least ε
  - Worst-case:
    \[ O(b^{C*/\varepsilon}) \]

- Space complexity:
  - **Same as time complexity**

- Optimality:
  - Nodes expanded in order of increasing path cost.
  - **YES**, if complete.
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)
DF-search; evaluation

Completeness;

- *Does it always find a solution if one exists?*
- **NO**
  - *unless* search space is finite and no loops are possible.
**DF-search; evaluation**

- Completeness;
  - **NO** unless search space is finite.

- Time complexity; \( O(b^m) \)
  - **Terrible if** \( m \) is much larger than \( d \) (depth of optimal solution)
DF-search; evaluation

- Completeness;
  - **NO** unless search space is finite.

- Time complexity; \( O(b^m) \)

- Space complexity; \( O(bm + 1) \)
  - Backtracking search uses even less memory
    - One successor instead of all \( b \).
DF-search; evaluation

- Completeness;
  - NO unless search space is finite.
- Time complexity;  $O(b^m)$
- Space complexity;  $O(bm + 1)$
- Optimality; No
  - Same issues as completeness
Depth-limited search

- DF-search with depth limit $l$.
  - i.e. nodes at depth $l$ have no successors; when depth $l$ is reached, go back.
  - Problem knowledge can be used to set value for $l$
- Solves the infinite-path problem.
- If $l < d$ then incompleteness results.
- Time complexity: $O(b^l)$
- Space complexity: $O(bl)$
Depth-limited algorithm

function DEPTH-LIMITED-SEARCH(problem, limit) return a solution or failure/cutoff
    return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) return a solution or failure/cutoff
    cutoff_occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] == limit then return cutoff
    else for each 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

- What?
  - A general strategy to find best depth limit $l$.
    - Goal is found at depth $d$, the depth of the shallowest goal-node.
  - Often used in combination with DF-search

- Combines benefits of DF- and BF-search
Iterative deepening search

function ITERATIVE_DEEPENING_SEARCH(problem) return a solution or failure

inputs: problem

for depth ← 0 to ∞ do
  result ← DEPTH-LIMITED_SEARCH(problem, depth)
  if result ≠ cutoff then return result

/* Note that the result returned is either a solution or a failure. */
ID-search, example

- Limit=0
ID-search, example

- Limit=1
ID-search, example

- Limit=2
ID-search, example

- Limit=3
ID search, evaluation

- Completeness:
  - YES (no infinite paths)
ID search, evaluation

- Completeness:
  - YES (no infinite paths)

- Time complexity:
  - Algorithm seems costly due to repeated generation of certain states.

- Node generation:
  - level d: once
  - level d-1: 2
  - level d-2: 3
  - ...
  - level 2: d-1
  - level 1: d

\[ N(IDS) = (d)b + (d-1)b^2 + ... + (1)b^d \]

\[ N(BFS) = b + b^2 + ... + b^d + (b^{d+1} - b) \]

Num. Comparison for b=10 and d=5 solution at far right

\[ N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123450 \]

\[ N(BFS) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100 \]
ID search, evaluation

- Completeness:
  - YES (no infinite paths)
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$
ID search, evaluation

- Completeness:
  - YES (no infinite paths)
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$
- Optimality:
  - YES if step cost is 1.
Bidirectional search

- Two simultaneous searches from start and goal.
  - Motivation: \( b^{d/2} + b^{d/2} \neq b^d \)
- Check whether the node belongs to the other fringe before expansion.
- Complete and optimal if both searches are BF.
How to search backwards?

- To get final solution path, one must search backward
- The predecessor of each node should be efficiently computable.
  - When actions are easily reversible.
## Summary of algorithms

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-cost</th>
<th>Depth-First</th>
<th>Depth-limited</th>
<th>Iterative deepening</th>
<th>Bidirectional search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>YES(^a)</td>
<td>YES(^a,b)</td>
<td>NO</td>
<td>YES,</td>
<td>YES(^a)</td>
<td>YES(^a,d)</td>
</tr>
<tr>
<td>Time</td>
<td>(b^{d+1})</td>
<td>(b^{\lceil C/\varepsilon \rceil})</td>
<td>(b^m)</td>
<td>(b^l)</td>
<td>(b^d)</td>
<td>(b^{d/2})</td>
</tr>
<tr>
<td>Space</td>
<td>(b^{d+1})</td>
<td>(b^{\lceil C/\varepsilon \rceil})</td>
<td>(b^m)</td>
<td>(b^l)</td>
<td>(b^d)</td>
<td>(b^{d/2})</td>
</tr>
<tr>
<td>Optimal?</td>
<td>YES(^c)</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES(^c)</td>
<td>YES(^c,d)</td>
</tr>
</tbody>
</table>

- \(^a\) – complete if \(b\) is finite;
- \(^b\) – complete if step costs \(\geq \varepsilon\) for some positive \(\varepsilon\);
- \(^c\) – optimal if step costs are all identical;
- \(^d\) – if both directions use breadth-first search.
Repeated states

- Failure to detect repeated states can turn a solvable problems into unsolvable ones.
Graph search algorithm

- Closed list stores all expanded nodes

```plaintext
function GRAPHSERARCH(problem, fringe) return a solution or failure

closed ← an empty set
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)

loop do
  if EMPTY?(fringe) then return failure
  node ← REMOVE-FIRST(fringe)
  if GOAL-TEST[problem] applied to STATE[node] succeeds
    then return SOLUTION(node)
  if STATE[node] is not in closed then
    add STATE[node] to closed
    fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```
Graph search, evaluation

- Optimality:
  - GRAPH-SEARCH discard newly discovered paths.
    - This may result in a sub-optimal solution
    - YET: when uniform-cost search or BF-search with constant step cost is used, the Graph Search algorithm is optimal.

- Time and space complexity,
  - Proportional to the size of the state space
    (may be much smaller than $O(b^d)$ as required by Tree Search algorithm).
  - DF- and ID-search with closed list no longer has linear space requirements since all nodes are stored in closed list!!
Search with partial information

- Previous assumption:
  - Environment is fully observable
  - Environment is deterministic
  - Agent knows the effects of its actions

What if knowledge of states or actions is incomplete?
Search with partial information

- Partial knowledge of states and actions:
  - sensorless or conformant problem
    - Agent does not have sensors and does not know what the initial/current state is.
  - contingency problem
    - Environment is partially observable or actions are uncertain
    - Percepts provide new information after each action
    - If uncertainty is caused by actions of another agent: adversarial problem
  - exploration problem
    - When states and actions of the environment are unknown, agent must act to discover them.
Sensorless/Conformant problems

- start in \( \{1,2,3,4,5,6,7,8\} \)
  - e.g Right goes to \( \{2,4,6,8\} \). Solution?
    - \([\text{Right, Suck, Left, Suck}]\)
- When the world is not fully observable: reason about a set of states that might be reached
  - =belief state
Sensorless/Conformant problems

- Search space of belief states
- Goal state = belief state with all members as goal states.
- If $S$ states then $2^S$ possible belief states.
  - Not all of them are reachable and, hence, in the state space.
Belief states of vacuum-world
Contingency problems

- Contingency, start in \{1,3\}.
- Murphy’s law, Suck can dirty a clean carpet.
- Local sensing: location, dirt at current location only
  - Percept = [L,Dirty] -> \{1,3\}
  - [Suck] -> \{5,7\}
  - [Right] -> \{6,8\}
  - [Suck] in \{6\} -> \{8\} (Success)
  - BUT [Suck] in \{8\} -> failure

- Solution??
  - Belief-state: no fixed action sequence guarantees solution

- Relax requirement:
  - [Suck, Right, if [R,dirty] then Suck]
  - Select actions based on contingencies arising during execution.