Solving Problems by Searching

- Agents, Goal-Based Agents, Problem-Solving Agents
- Search Problems
- Blind Search Strategies

Definition. An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors or actuators.
Examples of Agents

- Human agents
- Robotic agents
- Software agents (or software robots or softbots).

How Agents Should Act?

The behavior of an agent depends on the following:

- The environment. This is the world where the agent lives.
- The percept sequence. This is the complete history of everything the agent has ever perceived.
  
  The agent can be described (almost completely) by an agent function that maps every given percept sequence to an action.
  
  The agent function will be implemented by an agent program.
- The performance measure. This is the objective criterion for success of an agent’s behavior. It is imposed by the agent designer. It might not be easy to define the performance measure.

Example: The performance measure of an automatic taxi driver should be ....
Goal-Based Agents

The behavior of an agent also depends on:

- The agent’s goals. A goal specifies what states of the environment are desirable for the agent.

Problem-Solving Agents

Problem-solving agents are a class of goal-based agents. Problem solving agents decide what to do by finding sequences of actions that lead to desirable states.

**Example:** Consider an agent in the city of Arad, Romania. How can it get to Bucharest the next day, on time for its flight?
Problem-Solving Agents

Problem-solving agents work by carrying out the following tasks repeatedly:

- **Goal formulation**: decide what the objective is.
- **Problem formulation**: decide what actions and states to consider in order to meet the objective.
- **Search**: find a sequence of actions that achieve the goal.
- **Execution**: execute the chosen sequence of actions.

Example: Route Finding in Romania
Our First Agent Program

function SIMPLEPROBLEMOSOLVINGAGENT(\texttt{percept}) returns an action

static seq, state, goal, problem

\begin{verbatim}
state ← UPDATESTATE(state, percept)
if seq is empty then
  goal ← FORMULATEGOAL(state)
  problem ← FORMULATEPROBLEM(state, goal)
  seq ← SEARCH(problem)
end
action ← FIRST(seq)
seq ← REST(seq)
return action
\end{verbatim}

Structure of Agents

Agent = Architecture + Program

The architecture makes the percepts from the sensors available to the program, runs the program, and feeds the program’s action choices to the effectors as they are generated.

We will only deal with agent programs in this course.
The basic elements of a search problem are:

- **The initial state.**

- The set of available actions. To specify the available actions we use a successor function \( \text{Succ} \) which, given a state \( x \), returns a set of ordered pairs \( (\text{action}, \text{successor state}) \). This set tells us what actions are possible in \( x \) and what states are reachable from \( x \) by executing these actions.

The initial state and the successor function define the state space of a search problem: the set of all states reachable from the initial state by any sequence of actions.

A path in the state space is any sequence of states connected by a sequence of actions.

---

The goal to be achieved. The goal is a set of world states called the goal states. Goals can be specified implicitly by a goal test i.e., a test which can be applied to a state to determine if it is a goal state.

- **A path cost** function is a function (usually denoted by \( g \)) which assigns a numeric cost to each path. The path cost will usually be the sum of the costs of the individual actions along the path.

  The step cost of taking action \( a \) to go from state \( x \) to state \( y \) is denoted by \( c(x, a, y) \).

A solution to a search problem is a path from the initial state to a goal state. A solution is optimal if it has the lowest path cost among all solutions.
Example: Route Finding in Romania

Example: Formulating as a Search Problem

The route finding problem from Arad to Bucharest can formally be specified as follows:

- The states specify the city we are in e.g., $\text{In}(\text{Arad})$.
- The only available action is $\text{GoTo}$ e.g., $\text{GoTo}(\text{Sibiu})$.
- For every city the successor function gives us a set of pairs $(\text{GoTo}(.), \text{In}(.))$. For example,
  
  $\text{Succ}(\text{Arad}) = \{(\text{GoTo}(\text{Sibiu}), \text{In}(\text{Sibiu})),$
  
  $(\text{GoTo}(\text{Timisoara}), \text{In}(\text{Timisoara})), (\text{GoTo}(\text{Zerind}), \text{In}(\text{Zerind}))\}$.
- The initial state is $\text{In}(\text{Arad})$. The goal state is $\text{In}(\text{Bucharest})$.
- The path cost can be the road distance in kilometers.
Formal specification:

- States: a state description specifies the location of each tile and blank
- Actions: blank moves L, R, U, D
- Goal state
- Path cost: length of the path
The 8-queens problem

Formal specification:

- States: any arrangement of 0 to 8 queens on board
- Actions: add a queen to any square
- Goal test: 8 queens on board, none attacked
- Path cost: zero

Size of state space: $64^8$
The 8-queens problem (cont’d)

Alternative specification:

- States: arrangements of 0 to 8 queens with none attacked
- Actions: place a queen in the left-most empty column such that it is not attacked by any other queen.
- Goal test: 8 queens on board, none attacked
- Path cost: zero

Size of state space: $8^8$

Search Problems in the Real World

- Route finding
- Touring problems e.g., travelling salesman
- Robot navigation
- Automatic assembly sequencing
- Protein design
- Query optimisation problems in DBMS
- Internet searching
- Automatic workflow generation
Almost all of the problems presented above have **NP-hard or worse** computational complexity.

Thus, we should not expect simple algorithms for search problems to be efficient. This is a big problem for search problems; we will try to find ways to tackle it!
Searching for Solutions (cont’d)

Comments:

• Finding a solution is done by searching through the state space. The trick is to maintain and extend a set of partial solutions.

• The choice of which state to expand next is determined by the search strategy.

• The search process is building up a search tree that is superimposed over the state space.

• It is important to distinguish between the state space and the search tree.

function TreeSearch(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
Search Tree Nodes

Nodes in the search tree can be represented by a data structure with five components:

- **State**: the state to which the node corresponds.
- **ParentNode**: the node in the search tree that generated this node.
- **Action**: the action that was applied to generate the node.
- **PathCost**: the cost of the path from the initial state to the node.
- **Depth**: the number of nodes on the path from the root to this node.

The Fringe or Frontier

The set of nodes awaiting to be expanded is called the **fringe** or **frontier**. It can be implemented as a **queue** with operations:

- **MAKEQUEUE(Elements)**
- **EMPTY?(Queue)**
- **REMOVEFRONT(Queue)**
- **QUEUINGFN(Elements, Queue)**
The General Tree Search Algorithm

function \textsc{TreeSearch}(\textit{problem}, \textit{QueuingFn}) 
returns a solution, or failure 

\textit{fringe} ← \textsc{MakeQueue}(\textsc{MakeNode}(\textit{InitialState}[\textit{problem}])) 

loop do 
  if \textit{fringe} is empty then return failure 
  \textit{node} ← \textsc{RemoveFront}(\textit{fringe}) 
  if \textsc{GoalTest}[\textit{problem}] applied to \textit{State}[\textit{node}] succeeds then 
    return \textit{node} 
  \textit{fringe} ← \textsc{QueuingFn}(\textsc{Expand}(\textit{node}, \textit{problem}), \textit{fringe}) 
end

The function \textsc{Expand} is responsible for calculating each of the components of the nodes it generates.

Search Algorithms

We will consider two kinds of search algorithms:

- Uninformed or blind
- Informed or heuristic

Evaluation criteria for a search algorithm:

- Completeness
- Optimality
- Time complexity
- Space complexity
Uninformed Search Methods

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
- Bidirectional search

Breadth-first search (BFS)

```plaintext
function BREADTHFIRSTSEARCH(problem)
returns a solution or failure
    return TREESEARCH(problem, ENQUEUEATEND)
```

Example:
Breadth-first search (cont’d)

Evaluation:
- **Complete?** Yes, if the branching factor $b$ is finite.
- **Time:** $O(b^{d+1})$ where $b$ is the branching factor and $d$ is the depth of the solution.
- **Space:** $O(b^{d+1})$. This is the biggest problem with BFS.
- **Optimal?** Yes, if all actions have identical non-negative costs.

Note: BFS finds the shallowest goal state.

Uniform-cost search (UCS)

Modifies BFS by always expanding the lowest cost node on the fringe (as measured by the path cost).

Example:
Uniform-cost search (cont’d)

Evaluation:

- **Complete?** Yes, under the assumptions given below.
- **Time:** $O(b^{\lceil C^*/\epsilon \rceil})$ where $b$ is the **branching factor**, $C^*$ is the cost of the optimal solution and every action costs at least $\epsilon > 0$.
- **Space:** same as time.
- **Optimal?** Yes, under the assumptions given below.

Completeness and optimality hold under the following assumptions:

- The branching factor is finite.
- The cost never decreases as we go along a path i.e., $g(\text{SUCCESSOR}(n)) \geq g(n)$ for every node $n$. This condition holds e.g., when each action costs at least $\epsilon > 0$.

If the second assumption holds, UCS **expands nodes in order of increasing path cost**. Thus the first goal node selected for expansion is the optimal solution.
Uniform-cost search (cont’d)

Notes:
- BFS is UCS with $g(n) = \text{Depth}(n)$.
- UCS works exactly like BFS when all action costs are the same $\epsilon > 0$.

We can see that UCS is close to Dijkstra’s algorithm for finding single-source shortest paths on directed graphs with arcs with non-negative weights.

Some differences:
- In UCS the search is stopped when a goal node is found.
- In UCS we may have several nodes of the same state, instead of several cost updates of a state.
- The time and space complexity of UCS is different because the parameters of interest are different.

Note the pruning done by UCS. We eliminate parts of the tree without having to examine them.
Depth-first search (DFS)

Depth-first search always expands one of the nodes at the **deepest level** of the search tree.

*Example:*

```
  *   
  /   
 /     
/       
/         
/           
/             
/               
/                 
/                   
/                     
/                       
```

Depth-first search (cont’d)

**function** `DEPTHFIRSTSEARCH(problem)`

**returns** a solution, or failure

`TREESearch(problem, ENQUEUENATFRONT)`
Depth-first search (cont’d)

Evaluation:

• Complete? No
• Time: $O(b^m)$ where $b$ is the branching factor and $m$ is the maximum depth of the search tree.
• Space: $O(bm)$.
• Optimal? No

Depth-limited search (DLS)

Like DFS but imposes a depth limit on search. E.g., for the “driving to Bucharest” example, a good depth-limit is 19 (we have 20 cities).

Evaluation:

• Complete? Yes, iff $l \geq d$ where $l$ is the depth limit and $d$ the depth of a solution.
• Time: $O(b^l)$
• Space: $O(bl)$
• Optimal? No

Question: Can we always find a good depth-limit?
Iterative-deepening search (IDS)

IDS sidesteps the issue of choosing the best depth-limit by trying all possible ones: 0,1,2 and so on.

```plaintext
function IterativeDeepeningSearch(problem)
returns a solution sequence
  for depth ← 0 to ∞ do
    if DepthLimitedSearch(problem, depth) succeeds then
      return its result
  end
return failure
```

Iterative-deepening search (cont’d)

Example:

```
<table>
<thead>
<tr>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

.....
Iterative-deepening search (cont’d)

**Question:** Is IDS wasteful?

**Answer:** No!

Let us assume that the solution is found when the last node at level $d$ is expanded. Then the number of nodes generated in a BFS to depth $d$ is

$$1 + b + b^2 + \cdots + b^d + (b^{d+1} - b)$$

The number of nodes generated in an IDS to depth $d$ is

$$(d + 1) + db + (d - 1)b^2 + \cdots + 2b^{d-1} + 1b^d$$

Using the above formulas we can see that BFS can actually be a lot more wasteful than IDS. For example, for $b = 10$ and $d = 5$, **BFS** generates 1,111,100 nodes and **IDS** 123,450 nodes.

---

**Evaluation:**

- **Complete?** Yes, under the assumptions for BFS.
- **Time:** $O(b^{d+1})$
- **Space:** $O(bd)$
- **Optimal?** Yes, under the assumptions for BFS.

**IDS** is the search algorithm of choice when the search space is large and the depth of the search is not known.
**Bidirectional search**

**Idea:** Search both forward from the initial state and backward from the goal. Stop when the two searches meet.

**Motivation:** $b^{d/2} + b^{d/2} < b^d$

**Problems:**
- What does it mean to search backwards from the goal?
- What if we have many possible goal states?
- Can we check efficiently that the two searches meet?
- What kind of search do we do in each half?

---

**Bidirectional search (cont’d)**

**Evaluation:**
- **Complete?** Yes, if the branching factor is finite and both directions use BFS.
- **Time:** $O(b^{d/2})$
- **Space:** $O(b^{d/2})$
- **Optimal?** Yes, if both directions use BFS and under the assumptions for BFS.
Avoiding Repeated States

Example:

Avoiding Repeated States (cont’d)

- In this case the state space is a graph.
- A solution is to avoid generating any state that was generated before. This can be enforced by keeping a list of the generated states called the closed list. In this case the fringe of unexpanded nodes is called the open list. The closed list can be implemented by a hash-table for retrieval in constant time. However, there is no easy way to avoid the space penalty!
The General Graph Search Algorithm

function GraphSearch(problem, QueuingFn)
returns a solution, or failure

closed ← an empty set

fringe ← MAKEQUEUE(MAKENODE(INITIALSTATE[problem]))

loop do
  if fringe is empty then return failure
  node ← REMOVEFRONT(fringe)
  if GOALTEST[problem] applied to STATE[node] succeeds then
    return node
  if STATE[node] is not in closed then
    add STATE[node] to closed
    fringe ← QUEUINGFN(EXPAND(node, problem), fringe)
  end

Summary

- Agents, Goal-Based Agents, Problem-Solving Agents
- Search Problems
- Blind Search Strategies

Readings: Chapter 3, Sections 3.1-3.5 of AIMA