Solving problems by searching

First Lecture Today (Thu 30 Jun)
Read Chapters 18.6.1-2, 20.3.1

Second Lecture Today (Thu 30 Jun)
Read Chapter 3.1-3.4

Next Lecture (Tue 5 Jul)
Chapters 3.5-3.7, 4.1-4.2

(Please read lecture topic material before and after each lecture on that topic)
You will be expected to know

• State-space search
  – Definitions of a problem and of a solution
  – State-space graph

• Tree-search (don’t remember visited nodes) vs. Graph-search (do remember them)

• States vs. nodes; node implementation

• Search strategy evaluation:
  – Complete? Time/space complexity? Optimal?
  – Parameters: b, d, m
Complete architectures for intelligence?

• Search?
  – Solve the problem of what to do.

• Logic and inference?
  – Reason about what to do.
  – Encoded knowledge/”expert” systems?
  – Know what to do.

• Learning?
  – Learn what to do.

• Modern view: It’s complex & multi-faceted.
Search?
Solve the problem of what to do.

• Formulate “What to do?” as a search problem.
  – Solution to the problem tells agent what to do.

• If no solution in the current search space?
  – Formulate and solve the problem of finding a search space that does contain a solution.
  – Solve original problem in the new search space.

• Many powerful extensions to these ideas.
  – Constraint satisfaction; means-ends analysis; planning; game playing; etc.

• Human problem-solving often looks like search.
Why Search?

• We are engaged in a bigger more important problem, and we hit a search sub-problem we need to solve.
  – We need to search in order to solve it and then get back quickly to what we really wanted to do in the first place.

• To predict the result of our actions in the future.

• There are many sequences of actions, each with its utility; we wish to maximize our performance measure.

• We wish only to achieve a goal; by any means at all.
• We wish to find the best (optimal) way to achieve it.
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 or choose next city
• **Find solution:**
  – sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest
Example: Romania
Environments Types

- **Static / Dynamic**
  Previous problem was static: no attention to changes in environment
- **Observable / Partially Observable / Unobservable**
  Previous problem was observable: it knew initial state.
- **Deterministic / Stochastic**
  Previous problem was deterministic: no new percepts were necessary, we can predict the future perfectly given our actions
- **Discrete / continuous**
  Previous problem was discrete: we can enumerate all possibilities
Why not Dijkstra’s Algorithm?

• Dijkstra’s algorithm inputs the entire graph.
  – We want to search in unknown spaces.
  – Essentially, we want to combine search with exploration.
  – E.g., an autonomous rover on Mars must search an unknown space.

• D’s algorithm takes connections as given.
  – We want to search based on agent’s actions, with unknown connections.
  – E.g., a Web-crawler may not know what further connections are available on an unexplored URL before visiting it.
  – E.g., the agent may not know the result of an action before trying it.

• D’s algorithm won’t work on infinite spaces.
  – We want to search in infinite spaces.
  – E.g., the logical reasoning space is infinite.
  – E.g., the real world is essentially infinite to a human-size agent.
Example: vacuum world

- Observable, start in #5. Solution?
Example: vacuum world

- Observable, start in #5.  
  Solution?  
  [Right, Suck]
Vacuum world state space graph
Example: vacuum world

- Unobservable, start in \{1,2,3,4,5,6,7,8\} e.g., Solution?
Example: vacuum world

- Unobservable, start in \{1, 2, 3, 4, 5, 6, 7, 8\} e.g., Solution?
  \[\text{[Right, Suck, Left, Suck]}\]
Problem Formulation

A problem is defined by five items:

1. **initial state** e.g., "at Arad"
2. **actions** $\text{Actions}(s) = \text{set of actions available in state } s$
3. **transition model** $\text{Result}(s,a) = \text{state that results from action } a \text{ in state } s$
   (alternative: successor function) $S(s) = \text{set of action–state pairs}$
e.g., $S(\text{Arad}) = \{<\text{Arad }\rightarrow \text{Zerind, Sibiu, Timisoara}>, \ldots\}$
4. **goal test**, e.g., $s = \"at Bucharest\", \text{Checkmate}(s)$
5. **path cost** (additive) e.g., sum of distances, number of actions executed, etc.
   – $c(x,a,y)$ is the **step cost**, assumed to be $\geq 0$, summed to yield path cost

A solution = sequence of actions leading from 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?** discrete: dirt and robot location
- **initial state?** any
- **actions?** Left, Right, Suck
- **goal test?** no dirt at all locations
- **path cost?** 1 per action
Example: 8-Queens

• **states?** -any arrangement of n<=8 queens
  -or arrangements of n<=8 queens in leftmost n columns, 1 per column, such that no queen attacks any other (BETTER!!).

• **initial state?** no queens on the board

• **actions?** -add queen to any empty square
  -or add queen to leftmost empty square, such that it is not attacked by other queens (BETTER!!).

• **goal test?** 8 queens on the board, none attacked.

• **path cost?** 1 per move (not really relevant)
Example: robotic assembly

- **states**: real-valued coordinates of robot joint angles parts of the object to be assembled
- **initial state**: rest configuration
- **actions**: continuous motions of robot joints
- **goal test**: complete assembly
- **path cost**: time to execute + energy used
Example: The 8-puzzle

- states?
- initial state?
- actions?
- goal test?
- path cost?

Try yourselves
Example: The 8-puzzle

• states? locations of tiles
• initial state? given
• actions? move blank left, right, up, down
• goal test? goal state (given)
• path cost? 1 per move

[Note: optimal solution of $n$-Puzzle family is NP-hard]
Tree search algorithms

• Basic idea:
  – Exploration of state space by generating successors of already-explored states (a.k.a. expanding states).
  – Every generated state is evaluated: *is it a goal state?*
Tree search example
Tree search example
Tree search example

Note that we come back to Arad often, wasting time & work

We will visit the same node often, wasting time & work

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
Repeated states

• Failure to detect repeated states can turn a linear problem into an exponential one!

• Test is often implemented as a hash table.

[Diagram showing repeated states and a tree structure]
Solutions to Repeated States

- **Graph search**
  - never generate a state generated before
    - must keep track of all possible states (uses a lot of memory)
    - e.g., 8-puzzle problem, we have $9! = 362,880$ states
    - approximation for DFS/DLS: only avoid states in its (limited) memory: avoid infinite loops by checking path back to root.
  - “visited?” test usually implemented as a hash table

State Space

Example of a Search Tree

faster, but memory inefficient
Implementation: states vs. nodes

- A **state** is a (representation of) a physical configuration.

- A **node** is a data structure constituting part of a search tree that contains information such as: state, parent node, action, path cost $g(x)$, depth.

- The **Expand function** creates new nodes, filling in the various fields and using the **SuccessorFn** of the problem to create the corresponding states.
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 $\infty$)
Summary

• Generate the search space by applying actions to the initial state and all further resulting states.

• Problem: initial state, actions, transition model, goal test, step/path cost

• Solution: sequence of actions to goal

• Tree-search (don’t remember visited nodes) vs. Graph-search (do remember them)

• Search strategy evaluation: b, d, m
  – Complete? Time? Space? Optimal?