Uninformed Search

Chapter 3

Some material adopted from notes by Charles R. Dyer, University of Wisconsin-Madison
Today’s topics

• Goal-based agents
• Representing states and actions
• Example problems
• Generic state-space search algorithm
• Specific algorithms
  – Breadth-first search
  – Depth-first search
  – Uniform cost search
  – Depth-first iterative deepening
• Example problems revisited
Big Idea

Allen Newell and Herb Simon developed the *problem space principle* as an AI approach in the late 60s/early 70s.

"The rational activity in which people engage to solve a problem can be described in terms of (1) a set of *states* of knowledge, (2) *operators* for changing one state into another, (3) *constraints* on applying operators, and (4) *control* knowledge for deciding which operator to apply next."

BTW

• **Herb Simon** was a polymath who contributed to economics, cognitive science, management, computer science and many other fields

• He was awarded a Nobel Prize in 1978 “for his pioneering research into the decision-making process within economic organizations”

• He is the only computer scientist to have won a Nobel Prize, although it was for his work in economics
Example: 8-Puzzle

Given an initial configuration of 8 numbered tiles on a 3x3 board, move the tiles to produce a desired goal configuration.
15 puzzle

• Popularized, but not invented, by Sam Loyd
• He offered $1000 to all who could solve it in 1896
• He sold many puzzles
• Its states form two disjoint spaces
• There was no path to solution from initial state he gave!

Sam Loyd's 1914 illustration of the unsolvable variation
Simpler: 3-Puzzle

Start:

```
3 2 1
2 3 1
```

Goal:

```
1 2 3
1 2 3
```

Start:

```
3 2 1
2 1 3
```

Goal:

```
1 2 3
1 2 3
```

Start:

```
3 2 1
2 3 1
```

Goal:

```
1 2 3
1 2 3
```

Start:

```
3 2 1
2 1 3
```

Goal:

```
1 2 3
1 2 3
```
Building goal-based agents

We must answer the following questions

– How do we represent the state of the “world”? 
– What is the goal and how can we recognize it? 
– What are the possible actions? 
– What relevant information do we encode to describe states, actions and their effects and thereby solve the problem?
Characteristics of 8-puzzle?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8-puzzle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Characteristics of 8-puzzle

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8-puzzle</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- This is typical of the problems worked on in the 60s and 70s
- And the algorithms for solving them a state-space search approach
Representing states

• State of an 8-puzzle?
Representing states

• State of an 8-puzzle?
• A 3x3 array of integer in \{0..8\}
• No integer appears twice
• 0 represents the empty space

• In Python, we might implement this using a nine-character string: “540681732”
• And write functions to map the 2D coordinates to a sting index
What’s the goal to be achieved?

• Describe situation we want to achieve, a set of properties that we want to hold, etc.
• Defining a **goal test** function that when applied to a state returns True or False
• For our problem:

```python
def isGoal(state):
    # return True iff state is a goal
    return state == "123405678"
```
What are the actions?

• *Primitive actions* for changing the state
  
  In a *deterministic* world: no uncertainty in an action’s effects (simple model)

• Given action and description of *current world state*, action completely specifies
  
  – Whether action *can* be applied to the current world (i.e., is it applicable and legal?) and
  
  – What state *results* after action is performed in the current world (i.e., no need for history information to compute the next state)
Representing actions

• Actions ideally considered as discrete events that occur at an instant of time

• Example, in a planning context
  – If state:inClass and perform action:goHome, then next state is state:atHome
  – There’s no time where you’re neither in class nor at home (i.e., in the state of “going home”)

Representing actions

• Actions for 8-puzzle?
Representing actions

• Actions for 8-puzzle?

• Number of actions/operators depends on the representation used in describing a state
  – Specify 4 potential moves for each of the 8 tiles, resulting in a total of \(4 \times 8 = 32\) actions
  – Or, specify four potential moves for “blank” square and we only need 4 actions

• A good representational can simplify a problem!
Representing states

• **Size of a problem** usually described in terms of possible **number of states**

• Examples*
  – Tic-Tac-Toe has about $3^9$ states ($19,683 \approx 2 \times 10^4$)
  – Checkers has about $10^{40}$ states
  – Rubik’s Cube has about $10^{19}$ states
  – Chess has about $10^{120}$ states in a typical game
  – Go has $2 \times 10^{170}$

• State space size $\approx$ solution difficulty

*these are rough upper-bounds, of course
Representing states

• Our estimates were loose upper bounds
• How many possible, legal states does tic-tac-toe really have?
• Simple upper bound: 9 board cells, each of which can be empty, O, or X: so \(3^9\) or 19,683
• Only 593 states remain after eliminating
  – impossible states
  – Rotations and reflections
Yes, examples include theorem proving and this simple example from Knuth (1964)

• Starting with the number 4, a sequence of square root, floor, and factorial operations can reach any desired positive integer

• To get to 5 from 4, do

\[ \text{floor} (\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} ) = 5 \]

• floor(sqrt (sqrt (sqrt (sqrt (sqrt (fact (fact 4))))))))
Infinitely hard to solve?

• No
• But you must be more careful in searching a space that may be infinite
• Some approaches (e.g., breadth first search) may be better than others (e.g., depth first search)
Some example problems

- Toy problems and microworlds
  - 8-Puzzle
  - Missionaries and Cannibals
  - Cryptarithmetic
  - Remove 5 Sticks
  - Water Jug Problem
- Real-world problems
- We’ll look at a few
The **8-Queens Puzzle**

- Place eight queens on a chessboard such that no queen attacks any other.
- We can generalize the problem to a NxN chessboard.
- What are the states, goal test, actions?

*Is this a solution?*
Route Planning

Find a route from Arad to Bucharest

A simplified map of major roads in Romania used in our text
Water Jug Problem

• Two jugs J1 & J2 with capacity C1 & C2
• Initially J1 has W1 water and J2 has W2 water
  – e.g.: full 5-gallon jug and empty 2-gallon jug
• Possible actions:
  – Pour from jug X to jug Y until X empty or Y full
  – Empty jug X onto the floor
• Goal: J1 has G1 water and J2 G2
  – G1 or G2 can be -1 to represent any amount
• E.g.: initially full jugs with capacities 3 and 1 liters, goal is to have 1 liter in each
So...

• How can we represent the states?
• What’s an initial state; how to recognize goal states
• What are the actions; how can we tell which can be done in a given state; what’s the resulting state
• How do we search for a solution from an initial state any goal state
• What is a solution, e.g.:
  – The goal state achieved, or
  – The path (i.e., sequence of actions) taking us from the initial state to a goal state?
Search in a state space

• Basic idea:
  – Create representation of initial state
  – Try all possible actions & connect states that result
  – Recursively apply process to the new states until we find a solution or are left with *dead ends*

• We need to keep track of the connections between states and might use a
  – *Tree* data structure or
  – *Graph* data structure

• A graph structure is best in general...
Search in a state space

Consider a water jug problem with a 3-liter and 1-liter jug, an initial state of (3,1) and a goal stage of (1,1).

The tree model of space shows the possible states at each step, while the graph model of space avoids redundancy and loops and is usually preferred.
Formalizing state space search

• A state space is a graph \((V, E)\) where \(V\) is a set of nodes and \(E\) is a set of arcs, and each arc is directed from a node to another node.

• **Nodes:** data structures with state description and other info, e.g., node’s parent, name of action that generated it from parent, etc.

• **Arcs:** instances of actions, head is a state, tail is the state that results from action, label on arc is action’s name or id.
Formalizing search in a state space

• Each arc has fixed, positive cost associated with it corresponding to the action cost
  – Simple case: all costs are 1

• Each node has a set of successor nodes produced by trying all legal actions that can be applied at node’s state
  – Expanding a node = generating its successor nodes and adding them and their associated arcs to the graph

• One or more nodes are marked as start nodes

• A goal test is applied to a state to determine if its associated node is a goal node
Example: Water Jug Problem

• Two jugs J1 and J2 with capacity C1 and C2
• Initially J1 has W1 water and J2 has W2 water
  – e.g.: a full 5-gallon jug and an empty 2-gallon jug
• Possible actions:
  – Pour from jug X to jug Y until X empty or Y full
  – Empty jug X onto the floor
• Goal: J1 has G1 water and J2 G2
  – G1 or G0 can be -1 to represent any amount
This is not a toy problem!

Clip from the 1995 film *Die Hard with a Vengeance*
Example: Water Jug Problem

Given full 5-gal. jug and empty 2-gal. jug, fill 2-gal. jug with one gallon

• State representation?
  – General state?
  – Initial state?
  – Goal state?

• Possible actions?
  – Condition?
  – Resulting state?

<table>
<thead>
<tr>
<th>Name</th>
<th>Cond.</th>
<th>Transition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty 5G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Empty 2G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2to5</td>
<td>x ≤ 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5to2</td>
<td>x ≥ 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5to2part</td>
<td>y &lt; 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Action table
Example: Water Jug Problem

Given full 5-gal. jug and empty 2-gal. jug, fill 2-gal jug with one gallon

- State = (x,y), where x is water in jug 1; y is water in jug 2
- Initial State = (5,0)
- Goal State = (-1,1), where -1 means any amount

<table>
<thead>
<tr>
<th>Name</th>
<th>Cond.</th>
<th>Transition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>dump1</td>
<td>x&gt;0</td>
<td>(x,y)→(0,y)</td>
<td>Empty Jug 1</td>
</tr>
<tr>
<td>dump2</td>
<td>y&gt;0</td>
<td>(x,y)→(x,0)</td>
<td>Empty Jug 2</td>
</tr>
<tr>
<td>pour_1_2</td>
<td>x&gt;0 &amp; y&lt;C2</td>
<td>(x,y)→(x-D,y+D) D = min(x,C2-y)</td>
<td>Pour from Jug 1 to Jug 2</td>
</tr>
<tr>
<td>pour_2_1</td>
<td>y&gt;0 &amp; X&lt;C1</td>
<td>(x,y)→(x+D,y-D) D = min(y,C1-x)</td>
<td>Pour from Jug 2 to Jug 1</td>
</tr>
</tbody>
</table>
Formalizing search

• **Solution**: sequence of actions associated with a path from a start node to a goal node

• **Solution cost**: sum of the arc costs on the solution path
  
  – If all arcs have same (unit) cost, then solution cost is length of solution (number of steps)

  – Algorithms generally require that arc costs cannot be negative (why?)
Formalizing search

• **State-space search**: searching through state space for solution by **making explicit** a portion of an **implicit** state-space graph to find a goal node
  – Can’t materializing whole space for large problems
  – Initially V={S}, where S is the start node, E={}  
  – On **expanding** S, its **successor nodes** are generated and added to V and associated **arcs added to E**
  – Process continues until a goal node is found

• Nodes represent a **partial solution** path (+ cost of partial solution path) from S to the node
  – From a node there may be many possible paths (and thus solutions) with this partial path as a prefix
State-space search algorithm

;; problem describes the start state, operators, goal test, and operator costs
;; queueing-function is a comparator function that ranks two states
;; general-search returns either a goal node or failure

function general-search (problem, QUEUEING-FUNCTION)
    nodes = MAKE-QUEUE(MAKE-NODE(problem.INITIAL-STATE))
    loop
        if EMPTY(nodes) then return "failure"
        node = REMOVE-FRONT(nodes)
        if problem.GOAL-TEST(node.STATE) succeeds
            then return node
        nodes = QUEUEING-FUNCTION(nodes, EXPAND(node, problem.OPERATORS))
    end

;; Note: The goal test is NOT done when nodes are generated
;; Note: This algorithm does not detect loops
Key procedures to be defined

• EXPAND
  – Generate a node’s successor nodes, adding them to the graph if not already there

• GOAL-TEST
  – Test if state satisfies all goal conditions

• QUEUEING-FUNCTION
  – Maintain ranked list of nodes that are candidates for expansion
  – Changing definition of the QUEUEING-FUNCTION leads to different search strategies: Which node to expand next
Bookkeeping

Typical node data structure includes:

• State at this node
• Parent node(s)
• Action(s) applied to get to this node
• Depth of this node (# of actions on shortest known path from initial state)
• Cost of path (sum of action costs on best path from initial state)
Some issues

• Search process constructs a search tree/graph, where
  – **root** is initial state and
  – **leaf nodes** are nodes
    • not yet expanded (i.e., in list “nodes”) or
    • having no successors (i.e., they’re **deadends** because no operators were applicable and yet they are not goals)

• Search tree may be infinite due to loops; even graph may be infinite for some problems

• Solution is a **path** or a **node**, depending on problem.
  – E.g., in cryptarithmetic return a node; in 8-puzzle, a path

• Changing definition of the QUEUEING-FUNCTION leads to different search strategies
Informed vs. uninformed search

Uninformed search strategies (blind search)
  – Use no information about likely direction of a goal
  – Methods: breadth-first, depth-first, depth-limited, uniform-cost, depth-first iterative deepening, bidirectional

Informed search strategies (heuristic search)
  – Use information about domain to (try to) (usually) head in the general direction of goal node(s)
  – Methods: hill climbing, best-first, greedy search, beam search, algorithm A, algorithm A*
Evaluating search strategies

• Completeness
  – Guarantees finding a solution whenever one exists

• Time complexity (worst or average case)
  – Usually measured by number of nodes expanded

• Space complexity
  – Usually measured by maximum size of graph/tree during the search

• Optimality (aka Admissibility)
  – If a solution is found, is it guaranteed to be an optimal one, i.e., one with minimum cost
Classic uninformed search methods

The four classic uninformed search methods
– Breadth first search (BFS)
– Depth first search (DFS)
– Uniform cost search (generalization of BFS)
– Iterative deepening (blend of DFS and BFS)

To which we can add another technique
– Bi-directional search (hack on BFS)
Example of uninformed search strategies

Consider this search space where S is the start node, G is the goal, and numbers are arc costs. Assume graph is not known in advance.
Breadth-First Search

ignore weights on arcs

Expanded node

Nodes list (aka Fringe)

{ S

S

{ A

A

{ B

B

{ C

C

{ D

D

{ E

E

{ G

G

• Typically don’t check if node is goal until we expand it (why?)
• Solution path found is S A G , cost 18
• # nodes expanded (including goal node) = 7

FIFO (queue)

Notation

G

G is node; 18 is cost of shortest known path from S
Breadth-First Search (BFS)

- Long time to find solutions with many steps: we must look at all shorter length possibilities first

  - Complete tree of depth \( d \) where nodes have \( b \) children has \( 1+b+b^2+...+b^d = \frac{(b^{(d+1)}-1)}{(b-1)} \) nodes = \( \Theta(b^d) \)

  - Tree with depth 12 & branching 10 > trillion nodes
  - If BFS expands 1000 nodes/sec and nodes uses 100 bytes, can take 35 years & uses 111TB of memory!

  + Always finds solution if one exists

  + Solution found is optimal
Breadth-First Search

• Enqueue nodes in **FIFO** (first-in, first-out) order

• **Complete**

• **Optimal** (i.e., admissible) finds shortest path, which is optimal if all operators have same cost

• **Exponential time and space complexity**, $O(b^d)$, where $d$ is depth of solution; $b$ is branching factor (i.e., # of children)

• **Long time to find long solutions** since we explore all shorter length possibilities first
Depth-First Search

Expanded node | Nodes list (aka fringe)
---|---
$S^0$ | \{ $S^0$ \}
$A^3$ | \{ $A^3$ $B^1$ $C^8$ \}
$D^6$ | \{ $D^6$ $E^{10}$ $G^{18}$ $B^1$ $C^8$ \}
$E^{10}$ | \{ $E^{10}$ $G^{18}$ $B^1$ $C^8$ \}
$G^{18}$ | \{ $B^1$ $C^8$ \}

Solution path found is $S$ $A$ $G$, cost 18
Number of nodes expanded (including goal node) = 5
Depth-First (DFS)

- Enqueue nodes on nodes in **LIFO** (last-in, first-out) order, i.e., use stack data structure to order nodes
- **May not terminate** w/o **depth bound**, i.e., ending search below fixed depth D (depth-limited search)
- **Not complete** (with or w/o cycle detection, with or w/o a cutoff depth)
- **Exponential time**, $O(b^d)$, but **linear space**, $O(bd)$
- Can find **long solutions quickly** if lucky (and **short solutions slowly** if unlucky!)
- On reaching dead-end, can only back up one level at a time even if problem occurs because of a bad choice at top of tree
Uniform-Cost Search (UCS)

- Enqueue nodes by path cost. i.e., let $g(n) = \text{cost of path from } start \text{ to current node } n$. Sort nodes by increasing value of $g(n)$.
- Aka Dijkstra’s Algorithm and similar to Branch and Bound Algorithm from operations research
- Complete (*)
- Optimal/Admissible (*)
  Depends on goal test being applied when node is removed from nodes list, not when its parent node is expanded & node first generated
- Exponential time and space complexity, $O(b^d)$
Uniform-Cost Search

Expanded node | Nodes list
--- | ---
$S^0$ | $\{ B^1, A^3, C^8 \}$
$B^1$ | $\{ A^3, C^8, G^{21} \}$
$A^3$ | $\{ D^6, C^8, E^{10}, G^{18}, G^{21} \}$
$D^6$ | $\{ C^8, E^{10}, G^{18}, G^{21} \}$
$C^8$ | $\{ E^{10}, G^{13}, G^{18}, G^{21} \}$
$E^{10}$ | $\{ G^{13}, G^{18}, G^{21} \}$
$G^{13}$ | $\{ G^{18}, G^{21} \}$

Solution path found is $S \ C \ G$, cost 13
Number of nodes expanded (including goal node) = 7
Depth-First Iterative Deepening (DFID)

• Do DFS to depth 0, then (if no solution) DFS to depth 1, etc.
• Often used with a tree search
• Complete
• Optimal/Admissible if all operators have unit cost, else finds shortest solution (like BFS)
• Time complexity a bit worse than BFS or DFS

  Nodes near top of search tree generated many times, but since almost all nodes are near tree bottom, worst case time complexity still exponential, O(b^d)
Depth-First Iterative Deepening (DFID)

• If branching factor is $b$ and solution is at depth $d$, then nodes at depth $d$ are generated once, nodes at depth $d-1$ are generated twice, etc.
  ─ Hence $b^d + 2b^{(d-1)} + ... + db \leq b^d / (1 - 1/b)^2 = O(b^d)$.
  ─ If $b=4$, worst case is $1.78 \times 4^d$, i.e., 78% more nodes searched than exist at depth $d$ (in worst case)

• **Linear space complexity**, $O(bd)$, like DFS

• Has advantages of BFS (completeness) and DFS (i.e., limited space, finds longer paths quickly)

• Preferred for **large state spaces** where solution depth is unknown
How they perform

• **Depth-First Search:**
  – 4 Expanded nodes: S A D E G
  – Solution found: S A G (cost 18)

• **Breadth-First Search:**
  – 7 Expanded nodes: S A B C D E G
  – Solution found: S A G (cost 18)

• **Uniform-Cost Search:**
  – 7 Expanded nodes: S A D B C E G
  – Solution found: S C G (cost 13)

  *Only uninformed search that worries about costs*

• **Iterative-Deepening Search:**
  – 10 nodes expanded: S S A B C S A D E G
  – Solution found: S A G (cost 18)
## Comparing Search Strategies

<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 (if applicable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
<td>$b^{dr/2}$</td>
</tr>
<tr>
<td>Space</td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$bm$</td>
<td>$bl$</td>
<td>$bd$</td>
<td>$b^{dr/2}$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Complete?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes, if $l \geq d$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Searching Backward from Goal

• Usually, a successor function is reversible
  – i.e., can generate a node’s predecessors in graph

• If we know a single goal (rather than a goal’s properties), we could search backward to the initial state

• It might be more efficient
  – Depends on whether the graph fans in or out
Bi-directional search

- Alternate searching from the start state toward the goal and from the goal state toward the start
- Stop when the frontiers intersect
- Works well only when there are unique start & goal states
- Requires ability to generate “predecessor” states
- Can (sometimes) lead to finding a solution more quickly
Summary

• Search in a problem space is at the heart of many AI systems
• Formalizing the search in terms of states, actions, and goals is key
• The simple “uninformed” algorithms we examined can be augmented to heuristics to improve them in various ways
• But for some problems, a simple algorithm is best