Uninformed and Informed search algorithms

Chapter 3, 4 (Sections 1–2, 4)

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:

\[ \text{QUEUEINGFN} = \text{put successors at end of queue} \]

Properties of breadth-first search

Complete?? Yes (if \( b \) is finite)

Time?? \( 1 + b + b^2 + b^3 + \ldots + b^d = O(b^d), \) i.e., exponential in \( d \)

Space?? \( O(b^d) \) (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 1MB/sec so 24hrs = 86GB.
Uniform-cost search

Expand least-cost unexpanded node

Implementation:

$$\text{QUEUEINGFN} = \text{insert in order of increasing path cost}$$
Properties of uniform-cost search

**Complete??** Yes, if step cost $\geq \epsilon$

**Time??** # of nodes with $g \leq$ cost of optimal solution

**Space??** # of nodes with $g \leq$ cost of optimal solution

**Optimal??** Yes

---

Depth-first search

Expand deepest unexpanded node

**Implementation:**

```
QUEUEINGFN = insert successors at front of queue
```

 AraK
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 $l$

Implementation:
  Nodes at depth $l$ have no successors
Iterative deepening search

\textbf{function} \textsc{Iterative-Deepening-Search}( \textit{problem} ) \textbf{returns} a solution sequence
\textbf{inputs}: \textit{problem}, a problem

\textbf{for} \textit{depth} $\leftarrow$ 0 \textbf{to} $\infty$ \textbf{do}
\textbf{result} $\leftarrow$ \textsc{Depth-Limited-Search}( \textit{problem}, \textit{depth} )
\textbf{if} \textit{result} $\neq$ cut off \textbf{then} return \textit{result} \\
\textbf{end}

Properties of iterative deepening search

\underline{Complete??} Yes

\underline{Time??} $(d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)$

\underline{Space??} $O(bd)$

\underline{Optimal??} Yes, if step cost = 1

Can be modified to explore uniform-cost tree
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.

Review: General search

function \texttt{GENERAL-SEARCH(} \texttt{problem, QUEUING-FN)} \texttt{returns} a solution, or failure

\texttt{nodes} ← \texttt{MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))}

loop do
  if \texttt{nodes} is empty then return failure
  \texttt{node} ← \texttt{REMOVE-FRONT(nodes)}
  if \texttt{GOAL-TEST[problem]} applied to \texttt{STATE(node)} succeeds then return \texttt{node}
  \texttt{nodes} ← \texttt{QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))}
end

A strategy is defined by picking the order of node expansion

♦ Best-first search, A* search, Heuristics
♦ Hill-climbing, Simulated annealing
**Best-first search**

Idea: use an *evaluation function* for each node
- estimate of "desirability"

⇒ Expand most desirable unexpanded node

**Implementation:**

\[
\text{QueueingFn} = \text{insert successors in decreasing order of desirability}
\]

Special cases:
- greedy search
- A* search

---

**Romania with step costs in km**

<table>
<thead>
<tr>
<th>Location</th>
<th>Distance to Bucharest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>366</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Craiova</td>
<td>160</td>
</tr>
<tr>
<td>Dobrota</td>
<td>242</td>
</tr>
<tr>
<td>Eforie</td>
<td>161</td>
</tr>
<tr>
<td>Fagaras</td>
<td>178</td>
</tr>
<tr>
<td>Giurgiu</td>
<td>77</td>
</tr>
<tr>
<td>Hirsova</td>
<td>151</td>
</tr>
<tr>
<td>Iasi</td>
<td>226</td>
</tr>
<tr>
<td>Lugoj</td>
<td>244</td>
</tr>
<tr>
<td>Mehadia</td>
<td>241</td>
</tr>
<tr>
<td>Neamt</td>
<td>234</td>
</tr>
<tr>
<td>Oradea</td>
<td>380</td>
</tr>
<tr>
<td>Pitei</td>
<td>98</td>
</tr>
<tr>
<td>Rimnicu Vilcea</td>
<td>193</td>
</tr>
<tr>
<td>Sibiu</td>
<td>253</td>
</tr>
<tr>
<td>Timisoara</td>
<td>329</td>
</tr>
<tr>
<td>Urziceni</td>
<td>80</td>
</tr>
<tr>
<td>Vasli</td>
<td>199</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
</tbody>
</table>

Straight-line distance to Bucharest:

- Arad: 366 km
- Bucharest: 0 km
- Craiova: 160 km
- Dobrota: 242 km
- Eforie: 161 km
- Fagaras: 178 km
- Giurgiu: 77 km
- Hirsova: 151 km
- Iasi: 226 km
- Lugoj: 244 km
- Mehadia: 241 km
- Neamt: 234 km
- Oradea: 380 km
- Pitei: 98 km
- Rimnicu Vilcea: 193 km
- Sibiu: 253 km
- Timisoara: 329 km
- Urziceni: 80 km
- Vasli: 199 km
- Zerind: 374 km
Greedy search

Evaluation function $h(n)$ (heuristic)
   - estimate of cost from $n$ to goal

E.g., $h_{SLD}(n) = \text{straight-line distance from } n \text{ to Bucharest}$

Greedy search expands the node that *appears* to be closest to goal

◊ Greedy Search Example

Properties of greedy search

Complete?? No–can get stuck in loops, e.g.,

Iasi → Neamt → Iasi → Neamt →

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space?? $O(b^m)$—keeps all nodes in memory

Optimal?? No
A* search

Idea: avoid expanding paths that are already expensive

Evaluation function \( f(n) = g(n) + h(n) \)

- \( g(n) \) = cost so far to reach \( n \)
- \( h(n) \) = estimated cost to goal from \( n \)
- \( f(n) \) = estimated total cost of path through \( n \) to goal

A* search uses an admissible heuristic
i.e., \( h(n) \leq h^*(n) \) where \( h^*(n) \) is the true cost from \( n \).

E.g., \( h_{SLD}(n) \) never overestimates the actual road distance

Theorem: A* search is optimal

Optimality of A* (standard proof)

Suppose some suboptimal goal \( G_2 \) has been generated and is in the queue. Let \( n \) be an unexpanded node on a shortest path to an optimal goal \( G_1 \).

\[
f(G_2) = g(G_2) \quad \text{since } h(G_2) = 0
\]
\[
> g(G_1) \quad \text{since } G_2 \text{ is suboptimal}
\]
\[
\geq f(n) \quad \text{since } h \text{ is admissible}
\]

Since \( f(G_2) > f(n) \), A* will never select \( G_2 \) for expansion
Optimality of $A^*$ (more useful)

**Lemma:** $A^*$ expands nodes in order of increasing $f$ value

Gradually adds “$f$-contours” of nodes (cf. breadth-first adds layers)

Contour $i$ has all nodes with $f = f_i$, where $f_i < f_{i+1}$

---

Properties of $A^*$

**Complete??** Yes, unless there are infinitely many nodes with $f \leq f(G)$

**Time??** Exponential in [relative error in $h \times$ length of soln.]

**Space??** Keeps all nodes in memory

**Optimal??** Yes—cannot expand $f_{i+1}$ until $f_i$ is finished
Proof of lemma: Pathmax

For some admissible heuristics, \( f \) may decrease along a path

E.g., suppose \( n' \) is a successor of \( n \)

\[
\begin{array}{c}
n \quad g=5 \quad h=4 \quad f=9 \\
1 \\
n' \quad g'=6 \quad h'=2 \quad f'=8
\end{array}
\]

But this throws away information!
\( f(n) = 9 \Rightarrow \text{true cost of a path through } n \text{ is } \geq 9 \)
Hence true cost of a path through \( n' \) is \( \geq 9 \) also

Pathmax modification to A*:  
Instead of \( f(n') = g(n') + h(n') \), use \( f(n') = \max(g(n') + h(n'), f(n)) \)

With pathmax, \( f \) is always nondecreasing along any path

Admissible heuristics

E.g., for the 8-puzzle:
\( h_1(n) = \text{number of misplaced tiles} \)
\( h_2(n) = \text{total Manhattan distance} \)
(i.e., no. of squares from desired location of each tile)

\[
\begin{array}{c}
\begin{array}{ccc}
5 & 4 & 0 \\
6 & 1 & 8
\end{array} & \begin{array}{ccc}
1 & 2 & 3 \\
8 & 4
\end{array} \\
\begin{array}{ccc}
7 & 3 & 2 \\
7 & 6 & 5
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Start State} \\
\text{Goal State}
\end{array}
\end{array}
\]

\[ h_1(S) = ?? \]
\[ h_2(S) = ?? \]
Admissible heuristics

E.g., for the 8-puzzle:

- $h_1(n) =$ number of misplaced tiles
- $h_2(n) =$ total Manhattan distance
  (i.e., no. of squares from desired location of each tile)

\[
\begin{array}{ccc}
5 & 4 & \text{(start state)} \\
6 & 1 & 8 \\
7 & 3 & 2 \\
\end{array}
\quad
\begin{array}{ccc}
1 & 2 & 3 \\
8 & 4 \\
7 & 6 & 5 \\
\end{array}
\]

\[
h_1(S) = 7
\]
\[
h_2(S) = 2 + 3 + 3 + 2 + 4 + 2 + 0 + 2 = 18
\]

Dominance

If $h_2(n) \geq h_1(n)$ for all $n$ (both admissible) then $h_2$ dominates $h_1$ and is better for search

Typical search costs:

- $d = 14$ IDS = 3,473,941 nodes
- $A^*(h_1) = 539$ nodes
- $A^*(h_2) = 113$ nodes

- $d = 14$ IDS = too many nodes
- $A^*(h_1) = 39,135$ nodes
- $A^*(h_2) = 1,641$ nodes
Relaxed problems

Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem.

If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then \( h_1(n) \) gives the shortest solution.

If the rules are relaxed so that a tile can move to any adjacent square, then \( h_2(n) \) gives the shortest solution.

For TSP: let path be any structure that connects all cities \( \implies \) minimum spanning tree heuristic.

Iterative improvement algorithms

In many optimization problems, path is irrelevant; the goal state itself is the solution.

Then state space = set of “complete” configurations; find optimal configuration, e.g., TSP
or, find configuration satisfying constraints, e.g., n-queens.

In such cases, can use iterative improvement algorithms; keep a single “current” state, try to improve it.

Constant space, suitable for online as well as offline search.
**Example: Travelling Salesperson Problem**

Find the shortest tour that visits each city exactly once

![Diagram](image)

**Example: \(n\)-queens**

Put \(n\) queens on an \(n \times n\) board with no two queens on the same row, column, or diagonal

![Diagram](image)
Hill-climbing (or gradient ascent/descent)

“Like climbing Everest in thick fog with amnesia”

```
function HILL-CLIMBING( problem ) returns a solution state
inputs: problem, a problem
local variables: current, a node
               next, a node
current ← Make-Node( Initial-State[problem] )
loop do
    next ← a highest-valued successor of current
    if Value[next] < Value[current] then return current
    current ← next
end
```

Simulated annealing

Idea: escape local maxima by allowing some “bad” moves
but gradually decrease their size and frequency

```
function SIMULATED-ANNEALING( problem, schedule ) returns a solution state
inputs: problem, a problem
        schedule, a mapping from time to “temperature”
local variables: current, a node
                next, a node
                T, a “temperature” controlling the probability of downward steps
current ← Make-Node( Initial-State[problem] )
for t ← 1 to ∞ do
    T ← schedule[t]
    if T=0 then return current
    next ← a randomly selected successor of current
    ΔE ← Value[next] - Value[current]
    if ΔE > 0 then current ← next
    else current ← next only with probability e^{ΔE/T}
```
Properties of simulated annealing

At fixed “temperature” $T$, state occupation probability reaches Boltzman distribution

$$p(x) = \alpha e^{-\frac{E(x)}{kT}}$$

$T$ decreased slowly enough $\implies$ always reach best state

Is this necessarily an interesting guarantee??

Devised by Metropolis et al., 1953, for physical process modelling

Widely used in VLSI layout, airline scheduling, etc.