Artificial Intelligence

Informed Search and Exploration

Readings: Chapter 4 of Russell & Norvig.
Example: $n$-queens
Put $n$ queens on an $n \times n$ board with no two queens on the same row, column, or diagonal.

- Case 1: Consider one (fixed) cell at a time
- Case 2: Consider one row at a time
- Case 3: Consider one queen at a time
### n-queens

- **Case 1:** Consider one (fixed) cell at a time
- **Case 2:** Consider one row at a time
- **Case 3:** Consider one queen at a time

<table>
<thead>
<tr>
<th></th>
<th>case 1</th>
<th>case 2</th>
<th>case 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Branching factor:</strong></td>
<td>2</td>
<td>$n$</td>
<td>$n^2$</td>
</tr>
<tr>
<td><strong>Maximal depth:</strong></td>
<td>$n^2$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td><strong>State space:</strong></td>
<td>$2^{n^2}$</td>
<td>$n^n$</td>
<td>$n^{2^n}$</td>
</tr>
</tbody>
</table>
function TREE-SEARCH(problem, fringe) returns a solution, or failure

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

loop do

  if fringe is empty then return failure

  node ← REMOVE-FRONT(fringe)

  if GOAL-TEST[problem](STATE(node)) return node

  fringe ← INSERTALL(EXPAND(node, problem), fringe)

A strategy is defined by picking the order of node expansion
## Uninformed Search Strategies

<table>
<thead>
<tr>
<th>Strategies</th>
<th>Time</th>
<th>Space</th>
<th>Complete?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-first Search</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Depth-first Search</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</td>
<td>No</td>
</tr>
<tr>
<td>Depth-limited Search</td>
<td>$O(b^l)$</td>
<td>$O(bl)$</td>
<td>No</td>
</tr>
<tr>
<td>Iterative Deepening Search</td>
<td>$O(b^d)$</td>
<td>$O(bd)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Uniform Cost Search</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
<td>Yes</td>
</tr>
</tbody>
</table>

where $b$ is the branching factor, $d$ is the depth of the shadowest solution, $m$ is the length of the longest path, $l$ is the limit set by the user.
Informed Search Strategies

- Uninformed search strategies look for solutions by systematically generating new states and checking each of them against the goal.

- This approach is very inefficient in most cases.

- Most successor states are “obviously” a bad choice.

- Such strategies do not know that because they have minimal problem-specific knowledge.

- **Informed** search strategies exploit problem-specific knowledge as much as possible to drive the search.

- They are almost always *more efficient* than uninformed searches and often also *optimal*. 
Informed Search Strategies

Main Idea

- Use the knowledge of the problem domain to build an evaluation function $f$.

- For every node $n$ in the search space, $f(n)$ quantifies the desirability of expanding $n$ in order to reach the goal.

- Then use the desirability value of the nodes in the fringe to decide which node to expand next.
Informed Search Strategies

\[ f \] is typically an *imperfect measure* of the goodness of the node. The right choice of nodes is not always the one suggested by \( f \).

It is possible to build a perfect evaluation function, which will always suggest the right choice.

How?

Why don’t we use perfect evaluation functions then?
Standard Assumptions on Search Spaces

- The cost of a node increases with the node’s depth.

- Transitions costs are non-negative and bounded below. That is, there is a $\delta > 0$ such that the cost of each transition is $\geq \delta$.

- Each node has only finitely-many successors.

Note: There are problems that do not satisfy one or more of these assumptions.
Best-First Search

- Idea: use an *evaluation function* for each node to estimate of “desirability”
- Strategy: Always expand most desirable unexpanded node
- **Implementation**: fringe is a priority queue sorted in decreasing order of desirability
- Special cases:
  - uniform-cost search
  - greedy search
  - A* search
Implementing Best-first Search

**function** BEST-FIRST-SEARCH( *problem*, Eval-FN) **returns** a solution sequence

**inputs:** *problem*, a problem

     Eval-Fn, an evaluation function


Queueing-Fn ← a function that orders nodes by Eval-FN

**return** GENERAL-SEARCH( *problem*, Queueing-Fn)

**function** GENERAL-SEARCH( *problem*, QUEUING-FN) **returns** a solution, or failure

    nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))

    loop do
      if nodes is empty then **return** failure
      node ← REMOVE-FRONT(nodes)
      if GOAL-TEST[problem] applied to STATE(node) succeeds then **return** node
      nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
    end
Best-first Search Strategies

- There is a *whole family* of best-first search strategies, each with a different evaluation function.

- Typically, strategies use estimates of the *cost* of reaching the goal and try to *minimize* it.

- Uniform Search also tries to minimize a cost measure.

- Is it a best-first search strategy?

- Not in spirit, because the evaluation function should incorporate a *cost estimate of going from the current state to the closest goal state*. 
Greedy Search

- Evaluation function $h(n)$ (heuristic) is an estimate of cost from $n$ to the closest goal. E.g., $h_{SLD}(n) =$ straight-line distance from $n$ to Bucharest.

- Greedy search expands the node that *appears* to be closest to the goal.
Greedy Search Example
Greedy Search Example
Greedy Search Example

![Graph Diagram]

- **Arad** 366
- **Fagaras** 176
- **Oradea** 380
- **Rimnicu Vilcea** 193
- **Sibiu**
- **Timisoara** 329
- **Zerind** 374
Greedy search example
Properties of Greedy Search

- Complete??
Properties of greedy search

- **Complete??** No—can get stuck in loops, e.g., with Oradea as goal, Iasi → Neamt → Iasi → Neamt → Complete in finite space with repeated-state checking

- **Time??**
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??**
Properties of Greedy Search

- **Complete??** No—can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt → Iasi → Neamt → Iasi
  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??**
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 $n$ to a goal.
  (Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal $G$.)
  E.g., $h_{SLD}(n)$ never overestimates the actual road distance
- Theorem: A* search is optimal if $h$ is admissible.
A* Search Example

\[ 366 = 0 + 366 \]
A* Search Example
A* Search Example
A* Search Example
A* Search Example
A* Search Example
If $h$ is admissible, $f(n)$ never overestimates the actual cost of the best solution through $n$.

Overestimates are dangerous (h values are shown)

The optimal path is never found! (or maybe after a long time)
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 $G$.

$$f(G_2) = g(G_2) \quad \text{since } h(G_2) = 0$$
$$> g(G') \quad \text{since } G_2 \text{ is suboptimal}$$
$$\geq f(n) \quad \text{since } h \text{ is admissible}$$

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??
Properties of A*  

- **Complete**?? Yes, unless there are infinitely many nodes with $f \leq f(G)$
- **Time**??
Properties of A*

- **Complete??** Yes, unless there are infinitely many nodes with $f \leq f(G')$
- **Time??** $O(f \times |\{n \mid f(n) \leq f(G')\}|)$ (exponential in general in terms of the length of solutions)
- **Space??**
Properties of A*

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

- **Time??** \( O(f \ast |\{n \mid f(n) \leq f(G)\}|) \) (exponential in general in terms of the length of solutions)

- **Space??** \( O(|\{n \mid f(n) \leq f(G)\}|) \)

- **Optimal??**
Properties of A*

- **Complete??** Yes, unless there are infinitely many nodes with $f \leq f(G)$
- **Time??** $O(f \ast |\{n \mid f(n) \leq f(G')\}|)$ (exponential in general in terms of the length of solutions)
- **Space??** $O(|\{n \mid f(n) \leq f(G')\}|)$
- **Optimal??** Yes—cannot expand $f_{i+1}$ until $f_i$ is finished.
  - A* expands all nodes with $f(n) < C^*$
  - A* expands some nodes with $f(n) = C^*$
  - A* expands no nodes with $f(n) > C^*$
A heuristic is **consistent** if

\[ h(n) \leq c(n, a, n') + h(n') \]

If \( h \) is consistent, we have

\[
\begin{align*}
    f(n') &= g(n') + h(n') \\
    &= g(n) + c(n, a, n') + h(n') \\
    &\geq g(n) + h(n) \\
    &= f(n)
\end{align*}
\]

I.e., \( f(n) \) is nondecreasing along any path.
Admissible Heuristics

For the 8-puzzle:

- $h_1(n) =$ number of misplaced tiles
- $h_2(n) =$ total Manhattan distance (i.e., number of squares from desired location of each tile)

Start State

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

Goal State

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

$h_1(S) =$??

$h_2(S) =$??
Admissible Heuristics

For the 8-puzzle:

- $h_1(n) =$ number of misplaced tiles
- $h_2(n) =$ total Manhattan distance (i.e., number of squares from desired location of each tile)

Start State

<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Goal State

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

$h_1(S) = ?? 7$
$h_2(S) = ?? 4+0+3+3+1+0+2+1 = 14$
Dominance

- Definition: If $h_2(n) \geq h_1(n)$ for all $n$ (both admissible) then $h_2$ dominates $h_1$.

- For 8-puzzle, $h_2$ indeed dominates $h_1$.
  - $h_1(n) = $ number of misplaced tiles
  - $h_2(n) = $ total Manhattan distance

- If $h_2$ dominates $h_1$, then $h_2$ is better for search.

- For 8-puzzle, search costs:
  
  \[
  d = 14 \quad \text{IDS} = 3,473,941 \text{ nodes (IDS = Interactive Deepening Search)}
  \]
  \[
  A^*(h_1) = 539 \text{ nodes}
  \]
  \[
  A^*(h_2) = 113 \text{ nodes}
  \]
  
  \[
  d = 24 \quad \text{IDS} \approx 54,000,000,000 \text{ nodes}
  \]
  \[
  A^*(h_1) = 39,135 \text{ nodes}
  \]
  \[
  A^*(h_2) = 1,641 \text{ nodes}
  \]
Optimality/Completeness of A* Search

If the problem is solvable, A* always finds an optimal solution when

- the standard assumptions are satisfied,
- the heuristic function is admissible.

A* is optimally efficient for any heuristic function $h$: No other optimal strategy expands fewer nodes than A* when using the same $h$. 
Complexity of A* Search

- **Worst-case time complexity:** still exponential ($O(b^d)$) unless the error in $h$ is bounded by the logarithm of the actual path cost. That is, unless

  $$|h(n) - h^*(n)| \leq O(\log h^*(n))$$

  where $h^*(n) =$ actual cost from $n$ to goal.

- **Worst-Case Space Complexity:** $O(b^m)$ as in greedy best-first.

- A* generally runs out of memory before running out of time. (Improvements: IDA*, SMA*).
IDA* and SMA*

- IDA* (Iteractive Deepening A*): Set a limit and store only those nodes $x$ whose $f(x)$ is under the limit. The limit is increased by some value if no goal is found.

- SMA* (Simplified Memory-bound A*): Work like A*; when the memory is full, drop the node with the highest $f$ value before adding a new node.
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.

Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem.
Relaxed Problems

Well-known example: traveling salesperson problem (TSP)
Find the shortest tour visiting all cities exactly once

Minimum spanning tree can be computed in $O(n^2)$
and is a lower bound on the shortest (open) tour
Local Search Algorithms

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

Define state space as a set of "complete" configurations; find *optimal* configuration, e.g., TSP or, find configuration satisfying constraints, e.g., timetable

State space = set of "complete" configurations.

In such cases, can use local search (or iterative improvement) algorithms; keep a single "current" state, try to improve it.

Constant space, suitable for online as well as offline search
Local Search Example: TSP

- TSP: Traveling Salesperson Problem
- Start with any complete tour, perform pairwise exchanges

For $n$ cities, each state has $n(n - 1)/2$ neighbors.
Local Search Example: $n$-queens

- Put $n$ queens on an $n \times n$ board with no two queens on the same row, column, or diagonal.
- Change the row of a queen in a given column to reduce the number of conflicts.

For $n$ queens, each state has $n(n-1)$ neighbors.
Local Search Example: 8-queens

Standard and Compact Representations:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
c = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
5 & 7 & 2 & 6 & 1 & 4 & 3 & 8 \\
-4 & -5 & 1 & -2 & 4 & 2 & 4 & 0 \\
6 & 9 & 5 & 10 & 6 & 10 & 10 & 16
\end{bmatrix}
\]

Operation: Switching two columns.
Neighbors of each state: \(8 \times 7/2 = 28\).
Hill-Climbing (or Gradient Descent)

"Like climbing Everest in thick fog with amnesia"

function Hill-Climbing(problem) return state
node: current, neighbor;
current := Make-Node(Initial-State(problem));
loop do
    neighbor := highest-value-successor(current)
    if (Value(neighbor) < Value(current))
        then return State(current)
    else current := neighbor
end loop
end function

The returned state is a local maximum state.
Performance of Hill-Climbing

- Quality of the solution
  Problem: depending on initial state, can get stuck on local maxima

- Time to get the solution
  In continuous spaces, problems may be slow to converge.
  Choose a good initial solution; find good ways to compute the cost function

Improvements: Simulated annealing, tabu search, etc.