Example: $n$-queens

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

State space: the board with 0 to $n$ queens.
Initial state: the empty board or a board with $n$ random queens.
Goal test: No two queens attack each other.
Successors: ?
Informed search strategies look for solutions by systematically generating new states and checking each of them against the goal. Such strategies do not know that because they have minimal problem-specific knowledge. Most successor states are "obviously" a bad choice. This approach is very inefficient in most cases.

Informed search strategies exploit problem-specific knowledge. They are almost always more efficient than uninformed searches and often also optimal.

Uninformed search strategies look for solutions by systematically generating new states and checking each of them against the goal. Such strategies do not know that because they have minimal problem-specific knowledge. Most successor states are "obviously" a bad choice. This approach is very inefficient in most cases.

<table>
<thead>
<tr>
<th>Case</th>
<th>State space:</th>
<th>Branching factor:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$u_1 u_2$</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$u_1 u_2$</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>$u_1 u_2$</td>
<td>2</td>
</tr>
</tbody>
</table>

How to define the successors:

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

Why don't we use perfect evaluation functions then?

How?

Which will always suggest the right choice.

It is possible to build a perfect evaluation function, $f$.

But it is typically an imperfect measure of the goodness of the node. The right choice of nodes is not always the one suggested by $f$.

Why don't we use perfect evaluation functions then?

Artificial Intelligence – p.5/49
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.
- Representation: Transition is a priority queue sorted in decreasing order of desirability.
- Strategy: Always expand most desirable unexpanded node.
- Special cases:
  - Greedy search: if \( f(n) = h(n) \) is the estimated cost of node \( n \) reaching the goal.
  - Uniform-cost search: if \( f(n) = g(n) \) is the cost of node \( n \) from the initial node.
  - A* search: if \( f(n) = g(n) + h(n) \) is the estimated cost of node \( n \) reaching the goal.

Artificial Intelligence – p.8/49

Best-First Search

- Idea: use an evaluation function for each node to estimate desirability.
- Implementation: fringe is a priority queue sorted in decreasing order of desirability.
- locker: Always expand most desirable unexpanded node.
- Note: There are problems that do not satisfy one or more of these assumptions.
- Each node has only finitely-many successors.
- Transition is \( \geq 0 \).
- There is a \( \delta > 0 \) such that the cost of each transition is non-negative and bounded below.
- The cost of a node increases with the node's depth.
Implementing Best-first Search

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.

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

Is it a best-first search strategy?

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.

Uniform Search also tries to minimize a cost measure.
Greedy Search

Evaluation function \( h(n) \) (heuristic) is an estimate of cost from \( n \) to the closest goal.

For the \( n \)-queen problem, \( h_q(n) = \text{the number of attacks between the queens} \).

For the 8-puzzle, \( h_8(n) = \text{the number of tiles out of space comparing to the goal} \).

Greedy search expands the node that appears to be closest to goal.
Properties of Greedy Search

Complete?? No—can get stuck in loops, complete in finite space with repeated-state checking
Time??
Properties of Greedy Search

Complete? No–can get stuck in loops, complete in finite space with repeated-state checking

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

Space? \( O(b^m) \) or \( O(bm) \)—keeps all nodes in memory

Optimal? No

Complete? Yes–can get stuck in loops, complete in finite space with repeated-state checking

Artificial Intelligence – p.15/49
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 \( h^* \) is the true cost from \( n \) to a goal.

Also require \( h(n) \geq 0 \), so \( h(G) = 0 \) for any goal \( G \).

Example:

**Theorem**: \( A^* \) search is optimal if \( h \) is admissible.

\[ \text{E.g., } h^8(n) \text{ never overestimates the actual road distance} \]

\[ \text{Also require } h(n) \geq 0, \text{ so } h(G) = 0 \text{ for any goal } G. \]

\( A^* \) search uses an **admissible** heuristic.

Evaluation function
\[ (u)f = \text{estimated total cost of path through } u \text{ to goal} \]
\[ (u)g = \text{estimated cost from } u \text{ to goal} \]
\[ (u)b = \text{cost so far to reach } u \]

\[ (u)h + (u)b = (u)f \]

Idea: avoid expanding paths that are already expensive.
A* Search with Admissible Heuristic

We say \( h \) is admissible if \( h(n) \) never overestimates the actual cost of \( n \) reaching the best solution.

Overestimates are dangerous (\( h \) values are shown)

Optimality of A* (standard proof)

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

\[
(f(G_2) = g(G_2) & h(G_2) = 0) \quad \text{since} \quad \text{\( h \) is admissible}
\]

\[
(u) f \quad \geq \quad (G_2) f
\]

\[
(g(G_2)) \quad < \quad (G_2) f
\]

Since \( f(G_2) > f(n) \), A* will never select \( G_2 \) for expansion.

The optimal path is never found!
Optimality of A* (more useful)

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

Contour $i$ has all nodes with $f_i = f_{i+1}$, where $f_{i+1} > f_i$ (layers).

Gradually adds "f-contours" of nodes (cf. breadth-first adds).

Properties of A*
Properties of $A^*$

**Complete??**
Yes, unless there are infinitely many nodes

**Time??**
$O\left(\left|\{ n \mid f(n) \leq f(G)\}\right|\right)$
(Exponential in general in terms of the length of solutions)

**Space??**
Yes, unless there are infinitely many nodes
Properties of $A^*$

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

Time? $O(f^* |\{n | f(n) \leq f(G)\}|)$ (exponential in general in terms of the length of solutions).

Space? $O(||\{n | f(n) \leq f(G)\}||)$

Optimal? Yes—cannot expand $f_{i+1}$ until $f_i$ is finished.

$A^*$ expands all nodes with $f(n) < C^*$ and some nodes with $f(n) = C^*$.

$A^*$ expands no nodes with $f(n) > C^*$. 

Artificial Intelligence – p.25/49
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)

For the 8-puzzle:

Admissible Heuristics
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.

Optimality/Completeness of A* Search

- A* is optimally efficient for any heuristic function \( h \).
- The heuristic function is admissible.
- The standard assumptions are satisfied.

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

- A* is optimally efficient
- The heuristic function is admissible
- The standard assumptions are satisfied

Artificial Intelligence – p.29/49

Optimality/Completeness of A* Search

- For 8-puzzle, search costs:
  - IDS = 54,000,000 nodes
  - A* = 3,473,941 nodes (IDS = Iterative Deepening Search)

- For 8-puzzle, \( h_2 \) indeed dominates \( h_1 \).
- Definition: If \( h_2(u) \geq \sum h_1(u) \) for all \( u \) (both admissible) then \( h_2 \) dominates \( h_1 \).

Artificial Intelligence – p.30/49
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.

Worst-case space complexity: O(b^m) as in greedy best-first search. IDA* (Iterative 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*, but value before adding a new node.

IDA* and SMA*

Artificial Intelligence – p.31/49

Time (Improvements: IDA*, SMA*): A* generally runs out of memory before running out of time. (u)* costs from u to goal.

Worst-case time complexity: still exponential (O(\log |(u)*|^q)) unless the error in h is bounded by the logarithm of the actual path cost.
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 \( h^1(u) \) gives the shortest solution. If the rules of the 8-puzzle are relaxed so that a tile can move to any adjacent square, then \( h^2(u) \) gives the shortest solution.

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

Local Search Algorithms

In many optimization problems, path is irrelevant; the goal is to find an optimal configuration satisfying constraints, e.g., TSP (Traveling Salesman Problem) or \( n \)-queen problem. 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:

- Define state space as a set of “complete” configurations.
- State itself is the solution. E.g., \( n \)-queen problem.
- In many optimization problems, path is irrelevant; the goal is to find an optimal configuration satisfying constraints, e.g., TSP (Traveling Salesman Problem) or \( n \)-queen problem.
Start with any complete tour, and exchange a pair of cities in the tour. For $u$ cities, each state has $u(u-1)/2$ neighbors.

TSP: Traveling Salesman Problem

Local Search Example: TSP
Local Search Example: n-queens

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

**Operations**:
- Change the row of a queen in a given column. Each state has \( n(n-1) \) neighbors.
- Change the row of a queen in a given column to its neighboring rows. Each state has \( 2n \) neighbors.
- Exchange the positions of the queens in two columns. Each state has \( n(n-1)/2 \) neighbors.

**Neighbors of each state**: \( 8 \times 7/2 = 28 \).

**Standard and Compact Representations**:

### Compact Representation:

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

### Operations:
- Switching two columns.
- Changing the row of a queen in a given column.
- Changing the row of a queen in a given column to its neighboring rows.
- Exchanging the positions of two columns.
Hill-Climbing (or Gradient Descent)

**Function Hill-Climbing**

```plaintext
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.
  - Improvements: Random restarts, Simulated annealing, Tabu search, etc.

Choose a good initial solution; find good ways to compute the cost function.
Simulated Annealing

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

Realize it by a random assignment like

\[
\text{current} := \text{neighbor} \text{ with } \text{Prob}(\exp(-v/T))
\]

where \( v < 0 \) and \( |v| \) is the "size"; \( T > 0 \) is the "frequency" (also called temperature).

How to implement such a random assignment:

Generate a random real number \( x \) in \([0..1]\).

If \( x < \frac{1}{\exp(v/T)} \), then do the assignment; otherwise, do nothing.

Simulated Annealing Algorithm

```plaintext
function Simulated-Annealing(problem, schedule)
    return state

node: current, neighbor; integer: t, T;

current := Make-Node(Initial-State(problem));

for t := 1 to MAX-ITERATION do
    T := schedule[t];
    if (T == 0) return State(current);
    neighbor := random-successor(current);
    if (Value(neighbor) < Value(current))
        then v := Value(neighbor) - Value(current);
        current := neighbor with Prob(exp(v/T));
    else current := neighbor;

end for
end function
```

Simulated Annealing
Properties of Simulated Annealing

The smaller $|v|$, the greater $e^{v/T}$; the greater $T$, the greater $e^{v/T}$.

At fixed "temperature" $T$, state occupation probability reaches Boltzmann distribution $p(x) = \alpha e^{-E(x)/kT}$.

$T$ decreased slowly enough $\Rightarrow$ always reach best state.

Is this necessarily an interesting guarantee?

Devised by Metropolis et al., 1953, for physical processes.

Widely used in VLSI layout, airline scheduling, etc.

Artificial Intelligence – p.43/49

A Typical Temperature Function

$T = \alpha T_0$, where $\alpha = 0.999$. It starts with $T_0 = 100$ and becomes 0 gradually.

Artificial Intelligence – p.44/49
Tabu Search

Tabu Search is a local search with a Tabu list of nodes that is made up of nodes previously evaluated.

Three Tabu Search Strategies

- Forbidding strategy: control what nodes enter the Tabu list
- Freeing strategy: control what node exits the Tabu list and when
- Short-term strategy: manage interplay between the forbidding strategy and freeing strategy to select trial solutions

Typical Performance of SA
 Parameters of Tabu Search

- Local search procedure
- Neighborhood structure
- Aspiration conditions
- Addition of a tabu move
- Maximum size of tabu list

- If length = 1, it's the simple case of local search
- If length is very big, it's like A* or SMA search
- The use of tabu list
- Allows non-improving solution to be accepted in order to escape from a local optimum
- For larger and more difficult problems (scheduling, quadratic assignment and vehicle routing), tabu search obtains solutions that rival and often surpass the best previously found by other approaches.

 Advantages of Tabu Search

- Stopping rule
- If length is very big, it's like A* or SMA search
- Maximum size of tabu list
- Addition of a tabu move
- Aspiration conditions
- Neighborhood structure
- Local search procedure

Artificial Intelligence – p.47/49
Disadvantages of Tabu Search

- Too many parameters to be determined
- Number of iterations could be very large
- Incomplete, global optimum may not be found, depends on parameter settings