“First, they do an on-line search”
Example: The 8-puzzle

1 2 3
8 4
7 6 5

1 2 3
4 5 6
7 8
Example: Route Planning

![Map showing route planning example with start and end points labeled.](image-url)
Example: N Queens

4 Queens
Example: N Queens

4 Queens
State-Space Search Problems

General problem:
Given a start state, find a path to a goal state
- Can test if a state is a goal
- Given a state, can generate its successor states

Variants:
- Find any path vs. a least-cost path
- Goal is completely specified, task is just to find the path
  - Route planning
- Path doesn’t matter, only finding the goal state
  - 8 puzzle, N queens
Tree Representation of 8-Puzzle Problem Space
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] applied to State(node) succeeds return node

  fringe ← InsertAll(Expand(node, problem), fringe)

end loop

fringe (= frontier in the textbook) is the set of all leaf nodes available for expansion
Implementation: general tree search

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] applied to State(node) succeeds return node
        fringe ← InsertAll(Expand(node, problem), fringe)
    end loop

function Expand(node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in Successor-Fn[problem](State[node]) do
        s ← a new Node
        Parent-Node[s] ← node; Action[s] ← action; State[s] ← result
        Path-Cost[s] ← Path-Cost[node] + Step-Cost(node, action, s)
        Depth[s] ← Depth[node] + 1
        add s to successors
    end for
    return successors
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.
   includes parent, children, depth, path cost \( g(x) \)
**States** do not have parents, children, depth, or path cost!
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
    includes parent, children, depth, path cost \( g(x) \)
States do not have parents, children, depth, or path cost!

The \texttt{Expand} function creates new nodes, filling in the various fields and using the \texttt{SuccessorFn} of the problem to create the corresponding states.
Search strategies

A 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/expanded
  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 $\infty$)
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:

*fringe* is a FIFO queue, i.e., new successors go at end
Breadth-first search

Expand shallowest unexpanded node

Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end
Breadth-first search

Expand shallowest unexpanded node

Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end
Breadth-first search

Expand shallowest unexpanded node

Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end
Properties of breadth-first search

Complete??
Properties of breadth-first search

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

Time??
Properties of breadth-first search

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

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

**Space??**
Properties of breadth-first search

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

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

**Space??** \( O(b^d) \) (keeps every node in memory)

**Optimal??**
Properties of breadth-first search

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

Time?? \( b + b^2 + b^3 + \ldots + b^d = O(b^d) \), 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 for BFS.

Example: \( b = 10, \) 10,000 nodes/sec, 1KB/node

\( d = 3 \) \( \Rightarrow \) 1000 nodes, 0.1 sec, 1MB

\( d = 5 \) \( \Rightarrow \) 100,000 nodes, 10 secs, 100 MB

\( d = 9 \) \( \Rightarrow \) \( 10^9 \) nodes, 31 hours, 1 TB
Uniform-cost search

Expand least-cost unexpanded node (used when step costs are unequal)

Implementation:

\[ fringe = \text{queue ordered by path cost} \quad (\text{use priority queue}) \]

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost \( \geq \epsilon \) (small positive constant; 0 cost may cause infinite loop)

Time?? \# of nodes with \( g \leq \) cost of optimal solution, \( O(b^{[C^*/\epsilon]}) \)

where \( C^* \) is the cost of the optimal solution

Space?? \# of nodes with \( g \leq \) cost of optimal solution, \( O(b^{[C^*/\epsilon]}) \)

Optimal?? Yes—nodes expanded in increasing order of \( g(n) \)
Depth-first search

Expand deepest unexpanded node

Implementation: $\text{fringe} = \text{LIFO queue, i.e., put successors at front}$
Depth-first search

Expand deepest unexpanded node

Implementation:

\[ \text{fringe} = \text{LIFO queue, i.e., put successors at front} \]
Depth-first search

Expand deepest unexpanded node

Implementation:
\( fringe = \text{LIFO queue}, \text{i.e., put successors at front} \)
Depth-first search

Expand deepest unexpanded node

Implementation:

\[ \text{fringe} = \text{LIFO queue, i.e., put successors at front} \]
Depth-first search

Expand deepest unexpanded node

Implementation:

\[ \text{fringe} = \text{LIFO queue, i.e., put successors at front} \]
Depth-first search

Expand deepest unexpanded node

Implementation:

\[ fringe = \text{LIFO queue, i.e., put successors at front} \]
Depth-first search

Expand deepest unexpanded node

Implementation:

\( fringe = \) LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:

\[ \textit{fringe} = \text{LIFO queue, i.e., put successors at front} \]
Depth-first search

Expand deepest unexpanded node

**Implementation:**

\[fringe = \text{LIFO queue, i.e., put successors at front}\]
**Depth-first search**

Expand deepest unexpanded node

**Implementation:**

fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:

\[ \text{fringe} = \text{LIFO queue, i.e., put successors at front} \]
Depth-first search

Expand deepest unexpanded node

Implementation:

\[ \text{fringe} = \text{LIFO queue}, \text{i.e., put successors at front} \]
Properties of depth-first search

Complete??
Properties of depth-first search

**Complete??** No: fails in infinite-depth spaces, spaces with loops
   Modify to avoid repeated states along path ("GRAPH-SEARCH" in textbook)
   ⇒ complete in finite spaces

**Time??**
Properties of depth-first search

**Complete??** No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path (“GRAPH-SEARCH” in textbook)
⇒ complete in finite spaces

**Time??** \( O(b^m) \): terrible if \( m \) is much larger than \( d \) \( (m = \text{maximum depth}) \)
but if solutions are dense, may be much faster than breadth-first

**Space??**
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??**
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 (may find a solution but least cost solution may be on a different branch)
Depth-limited search

= depth-first search with depth limit \( l \),
i.e., nodes at depth \( l \) have no successors

Recursive implementation:

```plaintext
function Depth-Limited-Search(problem, limit) returns soln/fail/cutoff
    Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

function Recursive-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if Goal-Test[problem](State[node]) then return node
    else if Depth[node] = limit then return cutoff
    else for each successor in Expand(node, problem) do
        result ← Recursive-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```
Iterative deepening search

function \text{ITERATIVE-DEEPENING-SEARCH}(\textit{problem}) \textbf{returns} a solution

\hspace{1em} \textbf{inputs:} \textit{problem}, a problem

\hspace{2em} \textbf{for} depth \leftarrow 0 \textbf{ to } \infty \textbf{ do}

\hspace{3em} \textit{result} \leftarrow \text{DEPTH-LIMITED-SEARCH}(\textit{problem}, \textit{depth})

\hspace{3em} \textbf{if} \textit{result} \neq \textit{cutoff} \textbf{ then return} \textit{result}

\hspace{1em} \textbf{end}
Iterative deepening search \( l = 0 \)
Iterative deepening search $l = 1$

\[ \text{it} = 1 \]
Iterative deepening search $l = 2$
Iterative deepening search $l = 3$
Properties of iterative deepening search

Complete??
Properties of iterative deepening search

**Complete??** Yes

**Time??**
Properties of iterative deepening search

Complete?? Yes

Time?? \[ db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d) \]

Space??
Properties of iterative deepening search

Complete?? Yes
Time?? \( db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d) \)
Space?? \( O(bd) \)
Optimal??
Properties of iterative deepening search

**Complete??** Yes

**Time??** \[ db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d) \]

**Space??** \( O(bd) \)

**Optimal??** Yes, if step cost = 1
- Can be modified to explore uniform-cost tree
- Increasing path-cost limits instead of depth limits
- This is called *Iterative lengthening search* (exercise 3.17)
## Summary of algorithms

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes*</td>
<td>Yes*</td>
<td>No</td>
<td>Yes, if ( l \geq d )</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>( b^d )</td>
<td>( b^{[C^*/\epsilon]} )</td>
<td>( b^m )</td>
<td>( b^l )</td>
<td>( b^d )</td>
</tr>
<tr>
<td>Space</td>
<td>( b^d )</td>
<td>( b^{[C^*/\epsilon]} )</td>
<td>( b^m )</td>
<td>( b^l )</td>
<td>( b^d )</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes*</td>
<td>Yes*</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Forwards vs. Backwards

Problem: Find the shortest route
Bidirectional Search

Motivation: $b^{d/2} + b^{d/2} \ll b^d$

Can use breadth-first search or uniform-cost search

Hard for implicit goals e.g., goal = “checkmate” in chess
Repeated States

Failure to detect repeated states can turn a linear problem into an exponential one! (e.g., repeated states in 8 puzzle)

Graph search algorithm: Store expanded nodes in a set called closed (or explored) and only add new nodes to the fringe
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

    closed ← an empty set
    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]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
All these methods are slow (blind)

Can we do better?
Informed Search

Use problem-specific knowledge to guide search (use “heuristic function”)
**Best-first Search**

Generalization of breadth first search

Priority queue of nodes to be explored

Evaluation function $f(n)$ used for each node

- Insert initial state into priority queue
- While queue not empty
  - Node = head(queue)
  - If goal(node) then return node
  - Insert children of node into pr. queue
Who’s on (best) first?

Breadth first search is special case of best first
  • with \( f(n) = \text{depth}(n) \)

Dijkstra’s Algorithm is best first
  • with \( f(n) = g(n) \)
    where \( g(n) = \text{sum of edge costs from start to } n \)
Greedy best-first search

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

e.g., Route finding problems: $h_{SLD}(n) = \text{straight-line distance from } n \text{ to destination}$

Greedy best-first search expands the node that appears to be closest to goal
Example: Lost in Romania

Need: Shortest path from Arad to Bucharest
Example: Greedily Searching for Bucharest

\[ h_{SLD}(Arad) \]
Example: Greedily Searching for Bucharest
Example: Greedily Searching for Bucharest
Example: Greedily Searching for Bucharest

Not optimal!
Arad, Sibiu, Rimnicu Vilcea, Pitesti, Bucharest shorter

Greed doesn’t pay!
Properties of Greedy Best-First Search

Complete? No – can get stuck in loops (unless closed list is used)

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

Space? $O(b^m)$ -- keeps all nodes in memory a la breadth first search

Optimal? No, as our example illustrated
A* Search
(Hart, Nilsson & Rafael 1968)

• Best first search with \( f(n) = g(n) + h(n) \)

\( g(n) = \) sum of edge costs from start to \( n \)
\( h(n) = \) heuristic function = estimate of lowest cost path from \( n \) to goal

• If \( h(n) \) is “admissible” then search will be optimal

\( \begin{align*}
\text{Underestimates cost of any solution which can be reached from node}
\end{align*} \)
Back in Romania
Again

Aici noi
energie
iar!

Straight-line distance
to Bucharest
Arad 366
Bucharest 0
Craiova 160
Dobrota 242
Eforie 161
Fagaras 178
Giurgiu 77
Hirsova 151
Iasi 226
Lugoj 244
Mehadia 241
Neamt 234
Oradea 380
Pitesti 98
Rimnicu Vâlcea 193
Sibiu 253
Timisoara 329
Urziceni 80
Vaslui 199
Zerind 374
A* Example for Romania

\[ f(n) = g(n) + h(n) \] where

\[ g(n) = \text{sum of edge costs from start to } n \]

\[ h(n) = h_{SLD}(n) = \text{straight-line distance from } n \text{ to destination} \]
A* Example

- Sibiu: 393 = 140 + 253
- Timisoara: 447 = 118 + 329
- Zerind: 449 = 75 + 374
A* Example
A* Example

A* Example

Arad

Fagaras 415=239+176
Oradea 671=291+380

Sibiu

Timisoara 447=118+329
Zerind 449=75+374

Rimnicu Vilcea

Craiova 526=366+160
Pitesti 417=317+100
Sibiu 553=300+253

Arad 646=280+366
A* Example

- Arad
  - Sibiu
    - Arad: 646 = 280 + 366
    - Fagaras
    - Oradea: 671 = 291 + 380
    - Rimnicu Vilcea
      - Sibiu: 591 = 338 + 253
      - Bucharest: 450 = 450 + 0
  - Timisoara: 447 = 118 + 329
    - Zerind: 449 = 75 + 374
  - Craiova: 526 = 366 + 160
  - Pitesti: 417 = 317 + 100
  - Sibiu: 553 = 300 + 253
A* Example
Admissible heuristics

A heuristic $h(n)$ is admissible if
for every node $n$,

$$h(n) \leq h^*(n)$$

where $h^*(n)$ is the true cost to reach the goal state from $n$.

An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic.
Admissible Heuristics

Is the Straight Line Distance heuristic $h_{SLD}(n)$ admissible?
Admissible Heuristics

Is the Straight Line Distance heuristic $h_{SLD}(n)$ admissible?

Yes, it never overestimates the actual road distance

Theorem: If $h(n)$ is admissible, A* using TREE-SEARCH is optimal.
Suppose some suboptimal goal $G_2$ has been generated and is in the fringe. Let $n$ be an unexpanded node in the fringe such that $n$ is on a shortest path to an optimal goal $G$.

- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $> g(G)$ since $G_2$ is suboptimal
- $f(G) = g(G)$ since $h(G) = 0$
- $f(G_2) > f(G)$ from above
Suppose some suboptimal goal \( G_2 \) has been generated and is in the fringe. Let \( n \) be an unexpanded node in the fringe such that \( n \) is on a shortest path to an optimal goal \( G \).

\[
\begin{align*}
    f(G_2) &> f(G) \quad \text{from prev slide} \\
h(n) \leq h^*(n) &\quad \text{since } h \text{ is admissible} \\
g(n) + h(n) &\leq g(n) + h^*(n) \\
f(n) &\leq f(G) < f(G_2)
\end{align*}
\]

Hence \( f(n) < f(G_2) \Rightarrow A^* \text{ will never select } G_2 \text{ for expansion.} \)
Optimality of A*

A* expands nodes in order of increasing $f$ value
Gradually adds "$f$-contours" of nodes
Okay, proof is done!
Time to wake up...

AI rocks!
Properties of A*

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

Time? Exponential (for most heuristic functions in practice)

Space? Keeps all generated nodes in memory (exponential number of nodes)

Optimal? Yes
Admissible heuristics

E.g., for the 8-puzzle, what are some admissible heuristic functions? (for # steps to goal state)

\[ h_1(n) = ? \]

\[ h_2(n) = ? \]
Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n) = \text{number of misplaced tiles}$

$h_2(n) = \text{total Manhattan distance (no. of squares from desired location of each tile)}$

$h_1(S) = \ ?$

$h_2(S) = \ ?$
Admissible heuristics

E.g., for the 8-puzzle:

\( h_1(n) = \) number of misplaced tiles

\( h_2(n) = \) total Manhattan distance (no. of squares from desired location of each tile)

\[
\begin{align*}
    h_1(S) &= 8 \\
    h_2(S) &= 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18
\end{align*}
\]
Dominance

If $h_2(n) \geq h_1(n)$ for all $n$ (both admissible) then $h_2$ dominates $h_1$

$h_2$ is better for search
E.g., for 8-puzzle heuristics $h_1$ and $h_2$, typical search costs (average number of nodes expanded for solution depth $d$):

$d=12$  \[ \text{IDS} = 3,644,035 \text{ nodes} \]
  \[ A^*(h_1) = 227 \text{ nodes} \]
  \[ A^*(h_2) = 73 \text{ nodes} \]

$d=24$  \[ \text{IDS} = \text{too many nodes} \]
  \[ A^*(h_1) = 39,135 \text{ nodes} \]
  \[ A^*(h_2) = 1,641 \text{ nodes} \]
In general, A* not practical for large scale problems due to memory requirements (all generated nodes in memory)

Idea: Use iterative deepening
Iterative-Deepening A*

Like iterative-deepening search, but cutoff is f cost (= g + h) rather than depth

At each iteration, cutoff is smallest f cost among nodes that exceeded cutoff on prev iteration
Back to Admissible Heuristics

\[ f(x) = g(x) + h(x) \]

- \( g \): cost so far
- \( h \): underestimate of remaining costs

Where do heuristics come from?

e.g., \( h_{SLD} \)
Relaxed Problems

Derive admissible heuristic from exact cost of a solution to a relaxed version of problem

• For route planning, what is a relaxed problem?

Relax requirement that car stay on road ⇒ Straight Line Distance becomes optimal cost

Cost of optimal solution to relaxed problem ≤ cost of optimal solution for real problem
Heuristics for eight puzzle

What can we relax?

Original Problem: Tile can move from location A to B if A is horizontally or vertically next to B and B is blank
Heuristics for eight puzzle

Relaxed 1: Tile can move from any location A to any location B
Cost = $h_1 = \text{number of misplaced tiles}$

Relaxed 2: Tile can move from A to B if A is horizontally or vertically next to B (note: B does not have to be blank)
Cost = $h_2 = \text{total Manhattan distance}$

You can try other possible heuristics in your HW #1
Need for Better Heuristics

Performance of $h_2$ (Manhattan Distance Heuristic)

- 8 Puzzle  < 1 second
- 15 Puzzle  1 minute
- 24 Puzzle  65000 years

Can we do better?

Adapted from Richard Korf presentation
Creating New Heuristics

Given admissible heuristics $h_1, h_2, \ldots, h_m$, none of them dominating any other, how to choose the best?

Answer: No need to choose only one! Use:

$$h(n) = \max \{h_1(n), h_2(n), \ldots, h_n(n)\}$$

$h$ is admissible (why?)
$h$ dominates all $h_i$ (by construction)

Can we do better with:

$$h'(n) = h_1(n) + h_2(n) + \ldots + h_n(n)$$
Pattern Databases

Idea: Use solution cost of a subproblem as heuristic. For 8-puzzle: pick any subset of tiles

E.g., 3, 7, 11, 12

Precompute a table

• Compute optimal cost of solving just these tiles
  – This is a lower bound on actual cost with all tiles

• For all possible configurations of these tiles
  – Could be several million

• Use breadth first search back from goal state
  – State = position of just these tiles (& blank)

• Admissible heuristic $h_{DB}$ for complete state = cost of corresponding sub-problem state in database
Combining Multiple Databases

Can choose another set of tiles
  • Precompute multiple tables
How to combine table values?
  • Use the max trick!

E.g. Optimal solutions to Rubik’s cube
  • First found w/ IDA* using pattern DB heuristics
  • Multiple DBs were used (diff subsets of cubies)
  • Most problems solved optimally in 1 day
  • Compare with 574,000 years for IDS

Adapted from Richard Korf presentation
Drawbacks of Standard Pattern DBs

Since we can only take $\max$
  - Diminishing returns on additional DBs

Would like to be able to add values
  - But not exceed the actual solution cost (to ensure admissible heuristic)
  - How?
Disjoint Pattern DBs

Partition tiles into disjoint sets
  • For each set, precompute table
  • Don’t count moves of tiles not in set
    - This makes sure costs are disjoint
    - Can be added without overestimating!
    - E.g. For 15 puzzle shown, 8 tile DB has 519 million entries
    - And 7 tile DB has 58 million

During search
  • Look up costs for each set in DB
  • Add values to get heuristic function value

  • Manhattan distance is a special case of this idea where each set is a single tile

Adapted from Richard Korf presentation
Performance

15 Puzzle: 2000x speedup vs Manhattan dist
  - IDA* with the two DBs solves 15 Puzzle optimally in 30 milliseconds

24 Puzzle: 12 millionx speedup vs Manhattan
  - IDA* can solve random instances in 2 days.
  - Requires 4 DBs as shown
    - Each DB has 128 million entries
  - Without PDBs: 65000 years

Adapted from Richard Korf presentation
Next: Local Search

How to climb hills
How to reach the top by annealing
How to simulate and profit from evolution
Local search algorithms

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

Find configuration satisfying constraints, e.g., n-queens

In such cases, we can use local search algorithms

Keep a single "current" state, try to improve it
Example: \( n \)-queens

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

"Like climbing Everest in thick fog with amnesia"

**function** HILL-CLIMBING(problem) **returns** a state that is a local maximum
inputs: problem, a problem
local variables: current, a node
               neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
loop do
    neighbor ← a highest-valued successor of current
    if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
    current ← neighbor
Hill-climbing search

Problem: depending on initial state, can get stuck in local maxima
Example: 8-queens problem

Heuristic? (Value function)

\( h = \text{number of pairs of queens that are attacking each other, either directly or indirectly} \)

\( h = 17 \) for the above state (would like to minimize this)
Example: 8-queens problem

A local minimum with $h = 1$. Need $h = 0$

How to find global minimum (or maximum)?
Simulated Annealing

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

```plaintext
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 prob. 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^{Δ E / T}
```
Properties of simulated annealing

One can prove: If $T$ decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1.

Widely used in VLSI layout, airline scheduling, etc.
Local Beam Search

Keep track of $k$ states rather than just one

Start with $k$ randomly generated states

At each iteration, all the successors of all $k$ states are generated

If any one is a goal state, stop; else select the $k$ best successors from the complete list and repeat.
Hey, perhaps sex can improve search?
Sure – check out ye book.
Genetic Algorithms

A successor state is generated by combining two parent states

Start with $k$ randomly generated states (population)

A state is represented as a string over a finite alphabet (often a string of 0s and 1s)

Evaluation function (fitness function). Higher values for better states.

Produce the next generation of states by selection, crossover, and mutation
Example: 8-queens problem

Can we evolve a solution through genetic algorithms?

String Representation: 16257483
Example: Evolving 8 Queens

Sorry, wrong queens
Example: Evolving 8 Queens

Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 \times 7/2 = 28)

24/(24+23+20+11) = 31% probability of selection for reproduction

23/(24+23+20+11) = 29% etc
Queens crossing over

\[
\begin{array}{c}
32752411 \\
24748552 \\
\end{array}
\begin{array}{c}
32748552
\end{array}
\]

\[
\begin{array}{c}
\text{+}
\end{array}
\begin{array}{c}
\text{=}
\end{array}
\]

\[
\begin{array}{c}
\text{Queens crossing over}
\end{array}
\]
Let's move on to adversarial games
Adversarial Games

Programs that can play competitive board games

Minimax Search

Alpha-Beta Pruning
# Games Overview

<table>
<thead>
<tr>
<th>Perfect information</th>
<th>Deterministic</th>
<th>Chance</th>
</tr>
</thead>
<tbody>
<tr>
<td>chess, checkers, go, othello</td>
<td>backgammon, monopoly</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Imperfect information</th>
<th>Deterministic</th>
<th>Chance</th>
</tr>
</thead>
<tbody>
<tr>
<td>poker, bridge, scrabble</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
When there is *more than one agent*, the future is not easily predictable anymore for the agent.

In *competitive environments* (conflicting goals), adversarial search becomes necessary.

In AI, we usually consider special type of games:

- board games, which can be characterized as deterministic, turn-taking, two-player, zero-sum games with *perfect information*
Games as Search

Components:

- States:
- Initial state:
- Successor function:
- Terminal test:
- Utility function:
Games as Search

Components:

- **States**: board configurations
- **Initial state**: the board position and which player will move
- **Successor function**: returns list of \((move, state)\) pairs, each indicating a legal move and the resulting state
- **Terminal test**: determines when the game is over
- **Utility function**: gives a numeric value in terminal states (e.g., -1, 0, +1 in chess for loss, tie, win)
Games as Search

Convention: first player is called MAX, 2nd player is called MIN
MAX moves first and they take turns until game is over
Winner gets reward, loser gets penalty
Utility values stated from MAX's perspective
Initial state and legal moves define the game tree
MAX uses game tree to determine next move
Tic-Tac-Toe Example

MAX (X)

MIN (O)

MAX (X)

MIN (O)

TERMINAL

Utility

-1 0 +1
**Optimal Strategy: Minimax Search**

Find the contingent *strategy* for MAX assuming an infallible MIN opponent

Assumption: Both players play optimally!

Given a game tree, the optimal strategy can be determined by using the *minimax* value of each node (defined recursively):

\[
\text{MINIMAX-VALUE}(n) =
\begin{align*}
\text{UTILITY}(n) & \quad \text{If } n \text{ is a terminal} \\
\max_{s \in \text{succ}(n)} \text{MINIMAX-VALUE}(s) & \quad \text{If } n \text{ is a MAX node} \\
\min_{s \in \text{succ}(n)} \text{MINIMAX-VALUE}(s) & \quad \text{If } n \text{ is a MIN node}
\end{align*}
\]
Two-Ply Game Tree

“Ply” = move by 1 player
Two-Ply Game Tree

MAX

MIN

3 12 8 2 4 6 14 5 2
Two-Ply Game Tree
Two-Ply Game Tree

Minimax decision = \( A_1 \)

Minimax maximizes the worst-case outcome for max
What if MIN does not play optimally?

Definition of optimal play for MAX assumes MIN plays optimally

- Maximizes worst-case outcome for MAX

If MIN does not play optimally, MAX will do even better (i.e. at least as much or more utility obtained than if MIN was optimal) [Exercise 5.7 in textbook]
Another example
(4 ply)
Choose this move
Minimax Algorithm

function **MINIMAX-DECISION**(*state*) returns an action
  
v ← **MAX-VALUE**(*state*)
  return the action in **SUCCESSORS**(*state*) with value *v*

function **MAX-VALUE**(*state*) returns a utility value
  
  if **TERMINAL-TEST**(*state*) then return **UTILITY**(*state*)
  
v ← −∞
  for *a, s* in **SUCCESSORS**(*state*) do
    v ← **MAX**(*v, MIN-VALUE(*s*))
  return *v*

function **MIN-VALUE**(*state*) returns a utility value
  
  if **TERMINAL-TEST**(*state*) then return **UTILITY**(*state*)
  
v ← ∞
  for *a, s* in **SUCCESSORS**(*state*) do
    v ← **MIN**(*v, MAX-VALUE(*s*))
  return *v*
Properties of minimax

Complete? Yes (if tree is finite)

Optimal? Yes (against an optimal opponent)

Time complexity? $O(b^m)$

Space complexity? $O(bm)$ (depth-first exploration)
Good enough?

Chess:
- branching factor $b \approx 35$
- game length $m \approx 100$
- search space $b^m \approx 35^{100} \approx 10^{154}$

The Universe:
- number of atoms $\approx 10^{78}$
- age $\approx 10^{21}$ milliseconds

Can we search more efficiently?
Next Class:
Wrap up of search
Logic and Reasoning

To do:
Homework #1
Sign up for class mailing list