Announcements

- HW1 is due **Tuesday, January 30, 11:59 PM PT**
- Project 1 is due **Friday, February 2, 11:59 PM PT**

Pre-scan attendance QR code now!
(Password appears later)

[These slides were created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley. All CS188 materials are available at http://ai.berkeley.edu.]
[Updated slides from: Stuart Russell and Dawn Song]
A heuristic is:

- A function that *estimates* how close a state is to a goal
- Designed for a particular search problem
- Examples: Manhattan distance, Euclidean distance for pathing
Recap: Cost- vs. Heuristic-Guided Search

Uniform-Cost Search
(only costs, g)

Greedy Best-First Search
(only heuristic, h)

A* Search
(both, f=g+h)
Recap: Admissibility

Inadmissible (pessimistic) heuristics break optimality by trapping good plans on the fringe.

Admissible (optimistic) heuristics slow down bad plans but never outweigh true costs.
Recap: 8-Puzzle

Start State

Goal State
Designing a Heuristic: Knight’s moves

- Minimum number of knight’s moves to get from S to G?
  - $h_1 = \frac{\text{Manhattan distance}}{3}$
    - $h_1' = h_1$ rounded up to correct parity (even if S, G same color, odd otherwise)
  - $h_2 = \frac{\text{Euclidean distance}}{\sqrt{5}}$
    - $h_2' = h_2$ rounded up to correct parity
  - $h_3 = \frac{\text{maximum horizontal or vertical distance}}{2}$
    - $h_3' = h_3$ rounded up to correct parity
  - $h(n) = \max(h_1'(n), h_2'(n), h_3'(n))$ is admissible!
Recap: Optimality of A* Tree Search
Tree Search: Extra Work!

- Failure to detect repeated states can cause exponentially more work.
In BFS, for example, we shouldn’t bother expanding the circled nodes (why?)
Graph Search

- Idea: never expand a state twice

- How to implement:
  - Tree search + set of expanded states ("closed set")
  - Expand the search tree node-by-node, but...
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set

- Important: store the closed set as a set, not a list

- Can graph search wreck completeness? Why/why not?

- How about optimality?
A* Graph Search Gone Wrong?

State space graph

Search tree

Closed set

\{ S, B \}
A* Graph Search Gone Wrong?

State space graph

Search tree

Closed set

{ S B }

S (0+2)

SA (1+4)  SB (1+1)

SBC (3+1)  SBS (2+2)

SBCA (4+4)  SBCG (6+0)  SBCB (5+1)
A* Graph Search Gone Wrong?

State space graph

- **S**: Start node
- **A**: Node with 1 edge, h=4
- **B**: Node with 1 edge, h=2
- **C**: Node with 1 edge, h=1
- **G**: Goal node, h=0

Search tree

- **S**: Start node (0+2)
- **SA**: State A (1+4)
- **SB**: State B (1+1)
- **SAC**: State A with C (2+1)
- **SBC**: State B with C (3+1)
- **SBS**: State B with S (2+2)
- **SBCG**: State B with C with G (6+0)
- **SBCB**: State B with C with B (5+1)

Closed set

{ S B C A }
Consistency of Heuristics

- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal
    - $h(A) \leq \text{actual cost } h^* \text{ from } A \text{ to } G$
  - Consistency: heuristic “arc” cost ≤ actual cost for each arc
    - $h(A) - h(C) \leq \text{cost}(A \text{ to } C)$
      - a.k.a. “triangle inequality”: $h(A) \leq \text{cost}(A \text{ to } C) + h(C)$
      - Note: true cost $h^*$ necessarily satisfies triangle inequality

- Consequences of consistency:
  - The f value along a path never decreases
    - $h(A) \leq \text{cost}(A \text{ to } C) + h(C)$
  - A* graph search is optimal
A* Graph Search with Consistent Heuristic

State space graph

Search tree

Closed set

\{ S B A C \}
Consistency => non-decreasing f-score

Inconsistent

S (0+2) → SA (1+4) → SAC (2+1) → SBCA (4+4) → SBCG (6+0)

SB (1+1) → SBC (3+1) → SBCB (5+1)

SBS (2+2)

Consistent

S (0+2) → SA (1+2) → SAC (2+1) → SACB (4+1)

SB (1+1) → SBC (3+1) → SACG (5+0)

SBS (2+2)
Optimality of A* Graph Search

- Sketch: consider what A* does with a consistent heuristic:
  - Fact 1: In tree search, A* expands nodes in increasing total f value (f-contours)
  - Fact 2: For every state s, nodes that reach s optimally are expanded before nodes that reach s suboptimally
  - Result: A* graph search is optimal
Optimality

- Tree search:
  - A* is optimal if heuristic is admissible
  - UCS is a special case (h = 0)

- Graph search:
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)

- Consistency implies admissibility

- In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems
But...

- A* keeps the entire explored region in memory
- => will run out of space before you get bored waiting for the answer 😞
- There are variants that use less memory (Section 3.5.5):
  - IDA* works like iterative deepening, except it uses an $f$-limit instead of a depth limit
    - On each iteration, remember the smallest $f$-value that exceeds the current limit, use as new limit
    - Very inefficient when $f$ is real-valued and each node has a unique value
  - RBFS is a recursive depth-first search that uses an $f$-limit = the $f$-value of the best alternative path available from any ancestor of the current node
    - When the limit is exceeded, the recursion unwinds but remembers the best reachable $f$-value on that branch
  - SMA* uses *all available memory* for the queue, minimizing thrashing
    - When full, drop worst node on the queue but remember its value in the parent
Search and Models

- Search operates over models of the world
  - The agent doesn’t actually try all the plans out in the real world!
  - Planning is all “in simulation”
  - Your search is only as good as your models…
Search Gone Wrong?
Search Gone Wrong?
function Tree-Search(problem, fringe) return 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]) then return node
        for child-node in EXPAND(state[node], problem) do
            fringe ← Insert(child-node, fringe)
        end
    end
end
function GRAPH-SEARCH(problem, fringe) return 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 node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
        end
    end
Local Search

[These slides were created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley. All CS188 materials are available at http://ai.berkeley.edu.]
[Updated slides from: Stuart Russell and Dawn Song]
Local search algorithms

- In many optimization problems, *path* is irrelevant; the goal state *is* the solution.
- Then state space = set of “complete” configurations; find *configuration satisfying constraints*, e.g., n-queens problem; or, find *optimal configuration*, e.g., travelling salesperson problem.
- 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.
- More or less unavoidable if the “state” is yourself (i.e., learning).
Hill Climbing

- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit
Heuristic for $n$-queens problem

- Goal: $n$ queens on board with no \textit{conflicts}, i.e., no queen attacking another
- States: $n$ queens on board, one per column
- Actions: move a queen in its column
- Heuristic value function: number of conflicts
Hill-climbing algorithm

function HILL-CLIMBING(problem) returns a state
  current ← make-node(problem.initial-state)
loop do
  neighbor ← a highest-valued successor of current
  if neighbor.value ≤ current.value then
    return current.state
  current ← neighbor

“Like climbing Everest in thick fog with amnesia”
Global and local maxima

- Random restarts
  - find global optimum
  - duh

- Random sideways moves
  - Escape from shoulders
  - Loop forever on flat local maxima
Hill-climbing on the 8-queens problem

- No sideways moves:
  - Succeeds w/ prob. 0.14
  - Average number of moves per trial:
    - 4 when succeeding, 3 when getting stuck
  - Expected total number of moves needed:
    - $3(1-p)/p + 4 \approx 22$ moves

- Allowing 100 sideways moves:
  - Succeeds w/ prob. 0.94
  - Average number of moves per trial:
    - 21 when succeeding, 65 when getting stuck
  - Expected total number of moves needed:
    - $65(1-p)/p + 21 \approx 25$ moves

Moral: algorithms with knobs to twiddle are irritating
Simulated annealing

- Resembles the annealing process used to cool metals slowly to reach an ordered (low-energy) state

- Basic idea:
  - Allow “bad” moves occasionally, depending on “temperature”
  - High temperature => more bad moves allowed, shake the system out of its local minimum
  - Gradually reduce temperature according to some schedule
  - Sounds pretty flaky, doesn’t it?
Simulated annealing algorithm

function SIMULATED-ANNEALING(problem, schedule) returns a state

current ← problem.initial-state

for t = 1 to ∞ do
    T ← schedule(t)
    if T = 0 then return current
    next ← a randomly selected successor of current
    ∆E ← next.value - current.value
    if ∆E > 0 then current ← next
    else current ← next only with probability e^{ΔE/T}
**Simulated Annealing**

- **Theoretical guarantee:**
  - Stationary distribution (Boltzmann): \( P(x) \propto e^{E(x)/T} \)
  - If \( T \) decreased slowly enough, will converge to optimal state!

- **Proof sketch**
  - Consider two adjacent states \( x, y \) with \( E(y) > E(x) \) [high is good]
  - Assume \( x \rightarrow y \) and \( y \rightarrow x \) and outdegrees \( D(x) = D(y) = D \)
  - Let \( P(x), P(y) \) be the equilibrium occupancy probabilities at \( T \)
  - Let \( P(x \rightarrow y) \) be the probability that state \( x \) transitions to state \( y \)
Occupation probability as a function of $T$
Simulated Annealing

- Is this convergence an interesting guarantee?

- Sounds like magic, but reality is reality:
  - The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
  - “Slowly enough” may mean exponentially slowly
  - Random restart hillclimbing also converges to optimal state...

- Simulated annealing and its relatives are a key workhorse in VLSI layout and other optimal configuration problems
Local beam search

- **Basic idea:**
  - $K$ copies of a local search algorithm, initialized randomly
  - For each iteration
    - Generate ALL successors from $K$ current states
    - Choose best $K$ of these to be the new current states
  
  Or, $K$ chosen randomly with a bias towards good ones
Beam search example ($K=4$)
Local beam search

- Why is this different from $K$ local searches in parallel?
  - The searches *communicate*! “Come over here, the grass is greener!”
- What other well-known algorithm does this remind you of?
  - Evolution!
Genetic algorithms

- Genetic algorithms use a natural selection metaphor
  - Resample $K$ individuals at each step (selection) weighted by fitness function
  - Combine by pairwise crossover operators, plus mutation to give variety
Example: N-Queens

- Does crossover make sense here?
- What would mutation be?
- What would a good fitness function be?
Local search in continuous spaces
Example: Placing airports in Romania

Place 3 airports to minimize the sum of squared distances from each city to its nearest airport

Airport locations \[ x = (x_1, y_1), (x_2, y_2), (x_3, y_3) \]

City locations \( (x_c, y_c) \)

\( C_a = \) cities closest to airport \( a \)

Objective: minimize \[ f(x) = \sum_a \sum_{c \in C_a} (x_a - x_c)^2 + (y_a - y_c)^2 \]
Handling a continuous state/action space

1. Discretize it!
   - Define a grid with increment $\delta$, use any of the discrete algorithms

2. Choose random perturbations to the state
   a. First-choice hill-climbing: keep trying until something improves the state
   b. Simulated annealing

3. Compute gradient of $f(x)$ analytically
Finding extrema in continuous space

- Gradient vector $\nabla f(x) = (\partial f/\partial x_1, \partial f/\partial y_1, \partial f/\partial x_2, \ldots)^T$
- For the airports, $f(x) = \sum_a \sum_{c \in C_a} (x_a - x_c)^2 + (y_a - y_c)^2$
- $\partial f/\partial x_1 = \sum_{c \in C_1} 2(x_1 - x_c)$
- At an extremum, $\nabla f(x) = 0$
- Can sometimes solve in closed form: $x_1 = (\sum_{c \in C_1} x_c)/|C_1|$  
  - Is this a local or global minimum of $f$?
- If we can’t solve $\nabla f(x) = 0$ in closed form...
  - Gradient descent: $x \leftarrow x - \alpha \nabla f(x)$
- Huge range of algorithms for finding extrema using gradients
Many configuration and optimization problems can be formulated as local search

General families of algorithms:
- Hill-climbing, continuous optimization
- Simulated annealing (and other stochastic methods)
- Local beam search: multiple interaction searches
- Genetic algorithms: break and recombine states

Many machine learning algorithms are local searches