Final Review

• Please review your quizzes and old CS-171 tests
  • At least one question from a prior quiz or old CS-171 test will appear on the Final Exam (and all other tests)
### Final Review

<table>
<thead>
<tr>
<th>Week</th>
<th>Day Date</th>
<th>Quiz</th>
<th>Lecture 1 (1:00-2:20)</th>
<th>Lecture 2 (2:30-3:50)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Tue 5 Jul</td>
<td>Q2</td>
<td>Heuristic Search</td>
<td>Local Search</td>
</tr>
<tr>
<td></td>
<td>Thu 7 Jul</td>
<td></td>
<td>Mid-term Review</td>
<td>Mid-term Exam</td>
</tr>
<tr>
<td>4</td>
<td>Tue 12 Jul</td>
<td>Q3</td>
<td>Game Search A</td>
<td>Game Search B</td>
</tr>
<tr>
<td></td>
<td>Thu 14 Jul</td>
<td></td>
<td>Constraint Satisfaction A</td>
<td>Constraint Satisfaction B</td>
</tr>
<tr>
<td>5</td>
<td>Tue 19 Jul</td>
<td>Q4</td>
<td>Machine Learning A</td>
<td>Machine Learning B</td>
</tr>
<tr>
<td></td>
<td>Thu 21 Jul</td>
<td></td>
<td>Final Review</td>
<td>Final Review</td>
</tr>
<tr>
<td></td>
<td>Tue 26 Jul</td>
<td></td>
<td>Final Exam</td>
<td>Final Exam</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week</th>
<th>Day Date</th>
<th>Lecture 1 (1:00-2:20)</th>
<th>Lecture 2 (2:30-3:50)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Tue 5 Jul</td>
<td>Chapter 3.5-3.7</td>
<td>Chapter 4.1-4.2</td>
</tr>
<tr>
<td></td>
<td>Thu 7 Jul</td>
<td>Review all of the above</td>
<td>Mid-term Exam</td>
</tr>
<tr>
<td>4</td>
<td>Tue 12 Jul</td>
<td>Chapter 5.1, 5.2, 5.4</td>
<td>Chapter 5.3 (optional: 5.5+)</td>
</tr>
<tr>
<td></td>
<td>Thu 14 Jul</td>
<td>Chapter 6.1-6.4, except 6.3.3</td>
<td>same</td>
</tr>
<tr>
<td>5</td>
<td>Tue 19 Jul</td>
<td>Chapter 18.1-18.4</td>
<td>Chapters 18.5-12, 20.1-2</td>
</tr>
<tr>
<td></td>
<td>Thu 21 Jul</td>
<td>Review all of the above</td>
<td>Review all of the above</td>
</tr>
<tr>
<td></td>
<td>Tue 26 Jul</td>
<td>Final Exam (NO BREAK)</td>
<td>Final Exam</td>
</tr>
</tbody>
</table>

- Please review your quizzes and old CS-171 tests
- At least one question from a prior quiz or old CS-171 test will appear on the Final Exam (and all other tests)
You will be expected to know

- evaluation function \( f(n) \) and **heuristic function** \( h(n) \) for each node \( n \)
  - \( g(n) \) = known path cost so far to node \( n \).
  - \( h(n) \) = **estimate** of (optimal) cost to goal from node \( n \).
  - \( f(n) = g(n) + h(n) \) = **estimate** of total cost to goal through node \( n \).

- **Heuristic searches**: Greedy-best-first, A*
  - A* is optimal with admissible (tree)/consistent (graph) heuristics
  - Prove that A* is optimal with admissible heuristic for tree search
  - Recognize when a heuristic is admissible or consistent

- \( h_2 \) dominates \( h_1 \) iff \( h_2(n) \geq h_1(n) \) for all \( n \)
- Effective branching factor: \( b^* \)
- Inventing heuristics: relaxed problems; max or convex combination
Heuristic function (3.5)

- **Heuristic:**
  - Definition: a commonsense rule (or set of rules) intended to increase the probability of solving some problem
  - “using rules of thumb to find answers”

- **Heuristic function** $h(n)$
  - Estimate of (optimal) cost from $n$ to goal
  - Defined using only the *state* of node $n$
  - $h(n) = 0$ if $n$ is a goal node
  - Example: straight line distance from $n$ to Bucharest
    - Note that this is not the true state-space distance
    - It is an estimate — actual state-space distance can be higher

- Provides problem-specific knowledge to the search algorithm
Relationship of Search Algorithms

• $g(n)$ = known cost so far to reach $n$
• $h(n)$ = estimated (optimal) cost from $n$ to goal
• $f(n) = g(n) + h(n)$
  = estimated (optimal) total cost of path through $n$ to goal

• Uniform Cost Search sorts frontier by $g(n)$
• Greedy Best First Search sorts frontier by $h(n)$
• A* Search sorts frontier by $f(n)$
  – Optimal for admissible/consistent heuristics
  – Generally the preferred heuristic search
Greedy best-first search
(often called just “best-first”)

• \( h(n) \) = estimate of cost from \( n \) to \textit{goal}
  – e.g., \( h(n) \) = straight-line distance from \( n \) to Bucharest

• Greedy best-first search expands the node that \textbf{appears} to be closest to goal.
  – Sort queue by \( h(n) \)

• Not an optimal search strategy
  – May perform well in practice
Greedy best-first search example
Greedy best-first search example
Greedy best-first search example
Greedy best-first search example
Optimal Path

Values of $h_{SLD}$—straight-line distances to Bucharest.

<table>
<thead>
<tr>
<th>City</th>
<th>Distance</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>Drobeta</td>
<td>242</td>
</tr>
<tr>
<td>Eforie</td>
<td>161</td>
</tr>
<tr>
<td>Fagaras</td>
<td>176</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>Pitesti</td>
<td>100</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>Urzicieni</td>
<td>80</td>
</tr>
<tr>
<td>Vaslui</td>
<td>199</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
</tbody>
</table>
Greedy Best-first Search
With tree search, will become stuck in this loop

Order of node expansion:  S A D S A D S A D . . . .
Path found:  none      Cost of path found:  none  

![Diagram of the search space](https://via.placeholder.com/150)
- **D** with h=5
- **A** with h=7
- **B** with h=8
- **C** with h=9
- **S** with h=6
- **G** with h=0
Properties of greedy best-first search

• **Complete?**
  – Tree version can get stuck in loops.
  – Graph version is complete in finite spaces.

• **Time?** $O(b^m)$
  – A good heuristic can give **dramatic** improvement

• **Space?** $O(1)$ tree search, $O(b^m)$ graph search
  – Graph search keeps all nodes in memory
  – A good heuristic can give **dramatic** improvement

• **Optimal?** No
  – E.g., Arad $\rightarrow$ Sibiu $\rightarrow$ Rimnicu Vilcea $\rightarrow$ Pitesti $\rightarrow$ Bucharest is shorter!
A* search

• Idea: avoid paths that are already expensive
  – Generally the preferred simple heuristic search
  – Optimal if heuristic is:
    admissible (tree search)/consistent (graph search)

• Evaluation function $f(n) = g(n) + h(n)$
  – $g(n)$ = known path cost so far to node $n$.
  – $h(n)$ = estimate of (optimal) cost to goal from node $n$.
  – $f(n) = g(n)+h(n)$
    = estimate of total cost to goal through node $n$.

• *Priority queue sort function* = $f(n)$
A* tree search example

Values of $h(n)$—straight-line distances to Bucharest.
A* tree search example:
Simulated queue. City\(f=g+h\)

- Next:
- Children:
- Expanded:
- Frontier: \textbf{Arad}/366=0+366
A* tree search example:
Simulated queue. City/f=g+h

Arad/ 
366=0+366
A* tree search example:
Simulated queue. City/f=g+h

Arad/
366=0+366
A* tree search example: Simulated queue. City/f=g+h

- Next: Arad/366=0+366
- Children: Sibiu/393=140+253, Timisoara/447=118+329, Zerind/449=75+374
- Expanded: Arad/366=0+366
- Frontier: Arad/366=0+366, Sibiu/393=140+253, Timisoara/447=118+329, Zerind/449=75+374
A* tree search example:
Simulated queue. City/f=g+h

Sibiu/ 393=140+253
Timisoara/ 447=118+329
Zerind/ 449=75+374

Arad/ 366=0+366
A* tree search example:
Simulated queue. City/f=g+h
A* tree search example
A* tree search example: Simulated queue. City/f=g+h

- Next: Sibiu/393=140+253
- Expanded: Arad/366=0+366, Sibiu/393=140+253
A* tree search example:
Simulated queue. City/f=g+h
A* tree search example:
Simulated queue. City/f=g+h

- Sibiu/393=140+253
- Timisoara/447=118+329
- Zerind/449=75+374
- Arad/646=280+366
- Fagaras/415=239+176
- Oradea/671=291+380
- RimnicuVilcea/413=220+193

Arad/366=0+366
A* tree search example
A* tree search example:
Simulated queue. City/f=g+h

- Next: RimnicuVilcea/413=220+193
- Children: Craiova/526=366+160, Pitesti/417=317+100, Sibiu/553=300+253
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193
A* tree search example: Simulated queue. City/f=g+h
A* search example:
Simulated queue. City/f=g+h
A* tree search example

Note: The search below did not “back track.” Rather, both arms are being pursued in parallel on the queue.
A* tree search example:
Simulated queue. City/f=g+h

- Next: Fagaras/415=239+176
- Children: Bucharest/450=450+0, Sibiu/591=338+253
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193, Fagaras/415=239+176

Sibiu/553=300+253, Bucharest/450=450+0, Sibiu/591=338+253
A* tree search example

Note: The search below did not "back track." Rather, both arms are being pursued in parallel on the queue.
A* tree search example:
Simulated queue. City/f=g+h

- Next: Pitesti/417=317+100
- Children: Bucharest/418=418+0, Craiova/615=455+160, RimnicuVilcea/607=414+193
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193, Fagaras/415=239+176, Pitesti/417=317+100
A* tree search example
A* tree search example:
Simulated queue. City/f=g+h

- Next: Bucharest/418=418+0
- Children: **None; goal test succeeds.**
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193, Fagaras/415=239+176, Pitesti/417=317+100, Bucharest/418=418+0

Note that the short expensive path stays on the queue. The long cheap path is found and returned.
A* tree search example:
Simulated queue. City/f=g+h
A* tree search example:
Simulated queue. City/f=g+h
Properties of A*

• **Complete?** Yes
  (unless there are infinitely many nodes with \( f \leq f(G) \);
  can’t happen if step-cost \( \geq \varepsilon > 0 \))

• **Time/Space?** Exponential \( O(b^d) \)
  except if: \( |h(n) - h^*(n)| \leq O(\log h^*(n)) \)

• **Optimal?**
  (with: Tree-Search, admissible heuristic;
   Graph-Search, consistent heuristic)

• **Optimally Efficient?**
  (no optimal algorithm with same heuristic is guaranteed to expand fewer nodes)
Optimality of A* (proof)

Tree Search, where \( h(n) \) is admissible

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

**We want to prove:**

\[ f(n) < f(G_2) \]

(then A* will expand \( n \) before \( G_2 \))

- \( f(G_2) = g(G_2) \) since \( h(G_2) = 0 \)
- \( f(G) = g(G) \) since \( h(G) = 0 \)
- \( g(G_2) > g(G) \) since \( G_2 \) is suboptimal
- \( f(G_2) > f(G) \) from above, with \( h=0 \)
- \( h(n) \leq h^*(n) \) since \( h \) is admissible (under-estimate)
- \( g(n) + h(n) \leq g(n) + h^*(n) \) from above
- \( f(n) \leq f(G) \) since \( g(n)+h(n)=f(n) \) & \( g(n)+h^*(n)=f(G) \)
- \( f(n) < f(G_2) \) from above
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.
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance).
- Theorem: If $h(n)$ is admissible, A* using TREE-SEARCH is optimal.
Consistent heuristics
(consistent => admissible)

• A heuristic is **consistent** if for every node $n$, every successor $n'$ of $n$ generated by any action $a$,

$$h(n) \leq c(n, a, n') + h(n')$$

• If $h$ is consistent, we have

$$f(n') = g(n') + h(n') \leq g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

• i.e., $f(n)$ is non-decreasing along any path.

• **Theorem:**
  If $h(n)$ is consistent, $A^*$ using `GRAPH-SEARCH` is optimal

  keeps all checked nodes in memory to avoid repeated states
Dominance

• IF $h_2(n) \geq h_1(n)$ for all $n$
  THEN $h_2$ dominates $h_1$
  – $h_2$ is almost always better for search than $h_1$
  – $h_2$ guarantees to expand no more nodes than does $h_1$
  – $h_2$ almost always expands fewer nodes than does $h_1$
  – Not useful unless both $h_1$ & $h_2$ are admissible/consistent

• Typical 8-puzzle search costs
  (average number of nodes expanded):
  – $d=12$  
    IDS = 3,644,035 nodes
    $A^*(h_1) = 227$ nodes
    $A^*(h_2) = 73$ nodes
  – $d=24$  
    IDS = too many nodes
    $A^*(h_1) = 39,135$ nodes
    $A^*(h_2) = 1,641$ nodes
Effective branching factor: $b^*$

- Let $A^*$ generate $N$ nodes to find a goal at depth $d$
  - $b^*$ is the branching factor that a uniform tree of depth $d$ would have in order to contain $N+1$ nodes.
  \[
  N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d
  \]
  \[
  N + 1 = ((b^*)^{d+1} - 1) / (b^* - 1)
  \]
  \[
  N \approx (b^*)^d \Rightarrow b^* \approx d^{\frac{1}{d+N}}
  \]
  - For sufficiently hard problems, the measure $b^*$ usually is fairly constant across different problem instances.

- A good guide to the heuristic’s overall usefulness.
- A good way to compare different heuristics.
Inventing heuristics via “relaxed problems”

- A problem with fewer restrictions on the actions is called a relaxed problem.

- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original 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.

- Can be a useful way to generate heuristics.
  - E.g., ABSOLVER (Prieditis, 1993) discovered the first useful heuristic for the Rubik’s cube puzzle.
Please review your quizzes and old CS-171 tests

- At least one question from a prior quiz or old CS-171 test will appear on the Final Exam (and all other tests)

<table>
<thead>
<tr>
<th>Week</th>
<th>Day</th>
<th>Date</th>
<th>Quiz</th>
<th>Lecture 1 (1:00-2:20)</th>
<th>Lecture 2 (2:30-3:50)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Tue</td>
<td>5 Jul</td>
<td>Q2</td>
<td>Heuristic Search</td>
<td>Local Search</td>
</tr>
<tr>
<td></td>
<td>Thu</td>
<td>7 Jul</td>
<td></td>
<td>Mid-term Review</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Tue</td>
<td>12 Jul</td>
<td>Q3</td>
<td>Game Search A</td>
<td>Game Search B</td>
</tr>
<tr>
<td></td>
<td>Thu</td>
<td>14 Jul</td>
<td></td>
<td>Constraint Satisfaction A</td>
<td>Constraint Satisfaction B</td>
</tr>
<tr>
<td>5</td>
<td>Tue</td>
<td>19 Jul</td>
<td>Q4</td>
<td>Machine Learning A</td>
<td>Machine Learning B</td>
</tr>
<tr>
<td></td>
<td>Thu</td>
<td>21 Jul</td>
<td></td>
<td>Final Review</td>
<td>Final Review</td>
</tr>
<tr>
<td></td>
<td>Tue</td>
<td>26 Jul</td>
<td></td>
<td>Final Exam</td>
<td>Final Exam</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week</th>
<th>Day</th>
<th>Date</th>
<th>Lecture 1 (1:00-2:20)</th>
<th>Lecture 2 (2:30-3:50)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Tue</td>
<td>5 Jul</td>
<td>Chapter 3.5-3.7</td>
<td>Chapter 4.1-4.2</td>
</tr>
<tr>
<td></td>
<td>Thu</td>
<td>7 Jul</td>
<td>Review all of the above</td>
<td>Mid-term Exam</td>
</tr>
<tr>
<td>4</td>
<td>Tue</td>
<td>12 Jul</td>
<td>Chapter 5.1, 5.2, 5.4</td>
<td>Chapter 5.3 (optional: 5.5+)</td>
</tr>
<tr>
<td></td>
<td>Thu</td>
<td>14 Jul</td>
<td>Chapter 6.1-6.4, except 6.3.3</td>
<td>same</td>
</tr>
<tr>
<td>5</td>
<td>Tue</td>
<td>19 Jul</td>
<td>Chapter 18.1-18.4</td>
<td>Chapters 18.5-12, 20.1-2</td>
</tr>
<tr>
<td></td>
<td>Thu</td>
<td>21 Jul</td>
<td>Review all of the above</td>
<td>Review all of the above</td>
</tr>
<tr>
<td></td>
<td>Tue</td>
<td>26 Jul</td>
<td>Final Exam</td>
<td>Final Exam</td>
</tr>
</tbody>
</table>

(NO BREAK)
You will be expected to know

• Local Search Algorithms
  – Hill-climbing search
    • Gradient Descent in continuous spaces
  – Simulated annealing search
  – Genetic algorithms
  – (where applicable) Linear Programming

• Random Restart & Tabu Wrappers for above

• Difficulties: Local optima, plateaus, ridges, etc.

• Minimize cost, maximize value

  Value \approx \text{Constant} - \text{Cost}; \quad \text{Cost} \approx \text{Constant} - \text{Value}
Local search algorithms

- In many problems, the path to the goal is irrelevant; the goal state itself is the solution
  - **Local search**: Widely used for BIG problems
  - Returns good, but not optimal, solutions

- Search space = set of "complete" configurations
- Solution = configuration satisfying constraints
  - E.g., N-queens, VLSI layout, Airline flight scheduling

- Keep single "current" state, or small set of states.
  - Try to improve it or them.
- **Very memory efficient** (keep one or a few states)
  - You get to control how much memory you use.
Random Restart Wrapper

• These are stochastic local search methods
  – Different solution for each trial and initial state

• Almost every trial hits difficulties (see below)
  – Most trials will not yield a good result (sadly)

• Many random restarts improve your chances
  – Many “shots at goal” may, finally, get a good one

• Restart a random initial state; many times
  – Report the best result found; across many trials
Random Restart Wrapper

BestResultFoundSoFar <- infinitely bad;
UNTIL (you are tired of doing it) DO {
    Result <- (Local search from random initial state);
    IF (Result is better than BestResultFoundSoFar)
        THEN (Set BestResultFoundSoFar to Result);
}
RETURN BestResultFoundSoFar;

Typically, “you are tired of doing it” means that some resource limit is exceeded, e.g., number of iterations, wall clock time, CPU time, etc. It may also mean that Result improvements are small and infrequent, e.g., less than 0.1% Result improvement in the last week of run time.
Tabu Search Wrapper

• Recently visited states added to a tabu-list
  – Temporarily excluded from being visited again.

• Force solver away from explored regions
  – (In principle) avoids getting stuck in local minima.

• Implemented as Hash table + FIFO queue
  – Unit time cost per step; constant memory cost.

• You control how much memory is used
  – Run close to the edge but don’t blow out.
UNTIL ( you are tired of doing it ) DO {
    set Neighbor to makeNeighbor( CurrentState );
    IF ( Neighbor is in HASH ) THEN ( discard Neighbor );
    ELSE { push Neighbor onto FIFO, pop OldestState;
        remove OldestState from HASH, insert Neighbor;
        set CurrentState to Neighbor;
        run yourFavoriteLocalSearch on CurrentState; } }

Local Search Difficulties

These difficulties apply to ALL local search algorithms, and become MUCH more difficult as the dimensionality of the search space increases to high dimensions.

- **Problems:** depending on state, can get stuck in local maxima
  - Many other problems also endanger your success!!
Hill-climbing Difficulties

These difficulties apply to ALL local search algorithms, and become MUCH more difficult as the dimensionality of the search space increases to high dimensions.

- **Ridge problem**: Every neighbor appears to be downhill
  - But the search space has an uphill!! (worse in high dimensions)

**Ridge:**
Fold a piece of paper and hold it tilted up at an unfavorable angle to every possible search space step. Every step leads downhill; but the ridge leads uphill.
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: 8-queens problem

- $h =$ number of pairs of queens that attack each other, either directly or indirectly ($h = 17$ for this state)

Each number indicates $h$ if we move a queen in its column to that square

12 (boxed) = best $h$ among all neighbors; select one randomly
Hill-climbing search: 8-queens problem

A local minimum with $h = 1$

All one-step neighbors have higher $h$ values

What can you do to get out of this local minimum?
Simulated annealing search

- 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 $\leftarrow$ MAKE-NODE(INITIAL-STATE[problem])
    for $t \leftarrow 1$ to $\infty$ do
        $T \leftarrow$ schedule[$t$]
        if $T = 0$ then return current
        next $\leftarrow$ a randomly selected successor of current
        $\Delta E \leftarrow$ VALUE[next] − VALUE[current]
        if $\Delta E > 0$ then current $\leftarrow$ next
        else current $\leftarrow$ next only with probability $e^{\Delta E/T}$
```

Improvement: Track the **BestResultFoundSoFar**. Here, this slide follows Fig. 4.5 of the textbook, which is simplified.
P(accepting a worse successor)

- Decreases as Temperature $T$ decreases
- Increases as $|\Delta E|$ decreases

(Sometimes step size also decreases with $T$)

\[
e^\left(\frac{\Delta E}{T}\right)
\]

| $|\Delta E|$  | High  | Medium | Low  |
|--------------|-------|--------|------|
| High         | High  | Medium | Low  |
| Low          | High  | Medium | Medium |

```
next ← a randomly selected successor of current
\Delta E ← VALUE[next] - VALUE[current]
if $\Delta E > 0$ then current ← next
else current ← next only with probability $e^{\Delta E/T}$
```
Goal: “Ratchet” up a jagged slope
(see HW #2, prob. #5; here T = 1; cartoon is NOT to scale)

Value

A
Value=42

B
Value=41

C
Value=45

D
Value=44

E
Value=48

F
Value=47

G
Value=51

Arbitrary (Fictitious) Search Space Coordinate

Your “random restart wrapper” starts here.

You want to get here. HOW??

This is an illustrative cartoon.
Goal: “Ratchet” up a jagged slope
(see HW #2, prob. #5; here T = 1; cartoon is NOT to scale)

Your “random restart wrapper” starts here.

From A you will accept a move to B with \( P(AB) \approx 0.37 \).
From B you are equally likely to go to A or to C.
From C you are \( \approx 20 \times \) more likely to go to D than to B.
From D you are equally likely to go to C or to E.
From E you are \( \approx 20 \times \) more likely to go to F than to D.
From F you are equally likely to go to E or to G.
Remember best point you ever found (G or neighbor?).

<table>
<thead>
<tr>
<th>( x )</th>
<th>(-1)</th>
<th>(-4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e^x )</td>
<td>( \approx 0.37 )</td>
<td>( \approx 0.018 )</td>
</tr>
</tbody>
</table>
Genetic algorithms (Darwin!!)

• A state = a string over a finite alphabet (an individual)

• Start with $k$ randomly generated states (a population)

• **Fitness** function (= our heuristic objective function).
  – Higher fitness values for better states.

• **Select** individuals for next generation based on fitness
  – $P$(individual in next gen.) = individual fitness/$\Sigma$ population fitness

• **Crossover** fit parents to yield next generation (off-spring)

• **Mutate** the offspring randomly with some low probability
Fitness function: \#non-attacking queen pairs
- \( \text{min} = 0, \text{max} = 8 \times \frac{7}{2} = 28 \)
- \( \sum_i \text{fitness}_i = 24 + 23 + 20 + 11 = 78 \)
- \( P(\text{child}_1 \text{ in next gen.}) = \frac{\text{fitness}_1}{\sum_i \text{fitness}_i} = \frac{24}{78} = 31\% \)
- \( P(\text{child}_2 \text{ in next gen.}) = \frac{\text{fitness}_2}{\sum_i \text{fitness}_i} = \frac{23}{78} = 29\%; \) etc

How to convert a fitness value into a probability of being in the next generation.
Linear Programming

Efficient Optimal Solution
For a Restricted Class of Problems

Problems of the sort:

\[
\begin{align*}
\text{maximize } & \quad c^T x \\
\text{subject to : } & \quad A x \leq a; \quad B x = b
\end{align*}
\]

• Very efficient “off-the-shelves” solvers are available for LPs.

• They quickly solve large problems with thousands of variables.
Linear Programming Constraints

- Maximize: \( z = c_1 x_1 + c_2 x_2 + \ldots + c_n x_n \)
- Primary constraints: \( x_1 \geq 0, x_2 \geq 0, \ldots, x_n \geq 0 \)
- Arbitrary additional linear constraints:
  \[
  \begin{aligned}
  a_{i1} x_1 + a_{i2} x_2 + \ldots + a_{in} x_n &\leq a_i, (a_i \geq 0) \\
  a_{j1} x_1 + a_{j2} x_2 + \ldots + a_{jn} x_n &\geq a_j \geq 0 \\
  b_{k1} x_1 + b_{k2} x_2 + \ldots + b_{kn} x_n &= b_k \geq 0
  \end{aligned}
  \]

- Restricted class of linear problems.
  - Efficient for very large problems(!!) in this class.
Please review your quizzes and old CS-171 tests

At least one question from a prior quiz or old CS-171 test will appear on the Final Exam (and all other tests)
You Will Be Expected to Know

• Basic definitions (section 5.1)

• Minimax optimal game search (5.2)

• Evaluation functions (5.4.1)

• Cutting off search (5.4.2)

• Optional: Sections 5.4.3-4; 5.8
Games as Search

• Two players: MAX and MIN
• MAX moves first and they take turns until the game is over
  – Winner gets reward, loser gets penalty.
  – “Zero sum” means the sum of the reward and the penalty is a constant.

• Formal definition as a search problem:
  – **Initial state**: Set-up specified by the rules, e.g., initial board configuration of chess.
  – **Player(s)**: Defines which player has the move in a state.
  – **Actions(s)**: Returns the set of legal moves in a state.
  – **Result(s,a)**: Transition model defines the result of a move.
  – **Terminal-Test(s)**: Is the game finished? True if finished, false otherwise.
  – **Utility function(s,p)**: Gives numerical value of terminal state s for player p.
    • E.g., win (+1), lose (-1), and draw (0) in tic-tac-toe.
    • E.g., win (+1), lose (0), and draw (1/2) in chess.

• MAX uses search tree to determine “best” next move.
An optimal procedure: The Min-Max method

Will find the optimal strategy and best next move for Max:

1. Generate the whole game tree, down to the leaves.

2. Apply utility (payoff) function to each leaf.

3. Back-up values from leaves through branch nodes:
   - a Max node computes the Max of its child values
   - a Min node computes the Min of its child values

4. At root: Choose move leading to the child of highest value.
Two-Ply Game Tree

Minimax maximizes the utility of the worst-case outcome for Max

The minimax decision
Pseudocode for Minimax Algorithm

function MINIMAX-DECISION(state) returns an action
inputs: state, current state in game

return arg max_{a ∈ ACTIONS(state)} MIN-VALUE(Result(state, a))

function MIN-VALUE(state) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
v ← +∞
for a in ACTIONS(state) do
v ← MAX(v, MIN-VALUE(Result(state, a)))
return v

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

• **Complete?**
  – Yes (if tree is finite).

• **Optimal?**
  – Yes (against an optimal opponent).
  – Can it be beaten by an opponent playing sub-optimally?
    • No. (Why not?)

• **Time complexity?**
  – $O(b^m)$

• **Space complexity?**
  – $O(bm)$ (depth-first search, generate all actions at once)
  – $O(m)$ (backtracking search, generate actions one at a time)
Cutting off search

\texttt{MinimaxCutoff} is identical to \texttt{MinimaxValue} except

1. \texttt{Terminal?} is replaced by \texttt{Cutoff}?
2. \texttt{Utility} is replaced by \texttt{Eval}

Does it work in practice?

\[ b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4 \]

4-ply lookahead is a hopeless chess player!

4-ply \( \approx \) human novice
8-ply \( \approx \) typical PC, human master
12-ply \( \approx \) Deep Blue, Kasparov
Static (Heuristic) Evaluation Functions

• **An Evaluation Function:**
  – Estimates how good the current board configuration is for a player.
  – Typically, evaluate how good it is for the player, how good it is for the opponent, then subtract the opponent’s score from the player’s.
  – Othello: Number of white pieces - Number of black pieces
  – Chess: Value of all white pieces - Value of all black pieces

• Typical values from \(-\infty\) (loss) to \(+\infty\) (win) or \([-1, +1]\).

• If the board evaluation is \(X\) for a player, it’s \(-X\) for the opponent
  – “Zero-sum game”
For chess, typically \textit{linear} weighted sum of features

\[ \text{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) \]

\[ \text{e.g., } w_1 = 9 \text{ with } \]
\[ f_1(s) = (\text{number of white queens}) - (\text{number of black queens}) \text{, etc.} \]
Iterative (Progressive) Deepening

• In real games, there is usually a time limit \( T \) to make a move

• **How do we take this into account?**

• Using MiniMax we cannot use “partial” results with any confidence unless the full tree has been searched
  – So, we could be conservative and set a conservative depth-limit which guarantees that we will find a move in time \( < T \)
    • Disadvantage: we may finish early, could do more search

• In practice, Iterative Deepening Search (IDS) is used
  – IDS runs depth-first search with an increasing depth-limit
  – When the clock runs out we use the solution found at the previous depth limit
  – With alpha-beta pruning (next lecture), we can sort the nodes based on values found in previous depth limit to maximize pruning during the next depth limit => search deeper
Heuristics and Game Tree Search: Limited horizon

• The Horizon Effect
  – Sometimes there’s a major “effect” (e.g., a piece is captured) which is “just below” the depth to which the tree has been expanded
  – The computer cannot see that this major event could happen because it has a “limited horizon” --- i.e., when search stops
  – There are heuristics to try to follow certain branches more deeply to detect such important events, and so avoid stupid play
  – This helps to avoid catastrophic losses due to “short-sightedness”

• Heuristics for Tree Exploration
  – Often better to explore some branches more deeply in allotted time
  – Various heuristics exist to identify “promising” branches
  – Stop at “quiescent” positions --- all battles are over, things are quiet
  – Continue when things are in violent flux --- the middle of a battle
Selectively Deeper Game Trees

Diagram of a game tree with nodes labeled A, B, C, D, E, F, G, H, I, J, K, L, M, N, and O. The tree shows the possible moves and outcomes for both the computer and the opponents.
Eliminate Redundant Nodes

• On average, each board position appears in the search tree approximately $\sim 10^{150} / \sim 10^{40} \approx 10^{100}$ times.
  => Vastly redundant search effort.

• Can’t remember all nodes (too many).
  => Can’t eliminate all redundant nodes.

• However, some short move sequences provably lead to a redundant position.
  – These can be deleted dynamically with no memory cost

• Example:
  1. P-QR4 P-QR4;  2. P-KR4 P-KR4
  leads to the same position as
  1. P-QR4 P-KR4;  2. P-KR4 P-QR4
### Final Review

<table>
<thead>
<tr>
<th>Week</th>
<th>Day Date</th>
<th>Quiz</th>
<th>Lecture 1 (1:00-2:20)</th>
<th>Lecture 2 (2:30-3:50)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Tue 5 Jul</td>
<td>Q2</td>
<td>Heuristic Search</td>
<td>Local Search</td>
</tr>
<tr>
<td></td>
<td>Thu 7 Jul</td>
<td></td>
<td>Mid-term Review</td>
<td>Mid-term Exam</td>
</tr>
<tr>
<td>4</td>
<td>Tue 12 Jul</td>
<td>Q3</td>
<td>Game Search A</td>
<td>Game Search B</td>
</tr>
<tr>
<td></td>
<td>Thu 14 Jul</td>
<td></td>
<td>Constraint Satisfaction A</td>
<td>Constraint Satisfaction B</td>
</tr>
<tr>
<td>5</td>
<td>Tue 19 Jul</td>
<td>Q4</td>
<td>Machine Learning A</td>
<td>Machine Learning B</td>
</tr>
<tr>
<td></td>
<td>Thu 21 Jul</td>
<td></td>
<td>Final Review</td>
<td>Final Review</td>
</tr>
<tr>
<td></td>
<td>Tue 26 Jul</td>
<td></td>
<td>Final Exam</td>
<td>Final Exam</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week</th>
<th>Day Date</th>
<th>Lecture 1 (1:00-2:20)</th>
<th>Lecture 2 (2:30-3:50)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Tue 5 Jul</td>
<td>Chapter 3.5-3.7</td>
<td>Chapter 4.1-4.2</td>
</tr>
<tr>
<td></td>
<td>Thu 7 Jul</td>
<td>Review all of the above</td>
<td>Mid-term Exam</td>
</tr>
<tr>
<td>4</td>
<td>Tue 12 Jul</td>
<td>Chapter 5.1, 5.2, 5.4</td>
<td>Chapter 5.3 (optional: 5.5+)</td>
</tr>
<tr>
<td></td>
<td>Thu 14 Jul</td>
<td>Chapter 6.1-6.4, except 6.3.3</td>
<td>same</td>
</tr>
<tr>
<td>5</td>
<td>Tue 19 Jul</td>
<td>Chapter 18.1-18.4</td>
<td>Chapters 18.5-12, 20.1-2</td>
</tr>
<tr>
<td></td>
<td>Thu 21 Jul</td>
<td>Review all of the above</td>
<td>Review all of the above</td>
</tr>
<tr>
<td></td>
<td>Tue 26 Jul</td>
<td>Final Exam (NO BREAK)</td>
<td>Final Exam</td>
</tr>
</tbody>
</table>

- Please review your quizzes and old CS-171 tests
- At least one question from a prior quiz or old CS-171 test will appear on the Final Exam (and all other tests)
Alpha-beta Algorithm

- Depth first search
  - only considers nodes along a single path from root at any time

\[ \alpha = \text{highest-value choice found at any choice point of path for MAX} \]
  (initially, \( \alpha = -\infty \))

\[ \beta = \text{lowest-value choice found at any choice point of path for MIN} \]
  (initially, \( \beta = +\infty \))

- Pass current values of \( \alpha \) and \( \beta \) down to child nodes during search.
- Update values of \( \alpha \) and \( \beta \) during search:
  - MAX updates \( \alpha \) at MAX nodes
  - MIN updates \( \beta \) at MIN nodes
- Prune remaining branches at a node when \( \alpha \geq \beta \)
Pseudocode for Alpha-Beta Algorithm

function ALPHA-BETA-SEARCH(state) returns an action
inputs: state, current state in game

\[ v \leftarrow \text{MAX-VALUE}(state, -\infty, +\infty) \]

return the action in ACTIONS(state) with value \( v \)

function MAX-VALUE(state, \( \alpha \), \( \beta \)) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)

\[ v \leftarrow -\infty \]
for \( a \) in ACTIONS(state) do

\[ v \leftarrow \text{MAX}(v, \text{MIN-VALUE(Result(s,a), \( \alpha \), \( \beta \)))} \]
if \( v \geq \beta \) then return \( v \)
\[ \alpha \leftarrow \text{MAX}(\alpha, v) \]

return \( v \)

(MIN-VALUE is defined analogously)
When to Prune?

- Prune whenever $\alpha \geq \beta$.
  - Prune below a Max node whose alpha value becomes greater than or equal to the beta value of its ancestors.
    - *Max nodes update alpha* based on children’s returned values.
  - Prune below a Min node whose beta value becomes less than or equal to the alpha value of its ancestors.
    - *Min nodes update beta* based on children’s returned values.
α/β Pruning vs. Returned Node Value

• Some students are confused about the use of α/β pruning vs. the returned value of a node

• α/β are used ONLY FOR PRUNING
  – α/β have no effect on anything other than pruning
  – IF (α ≥ β) THEN prune & return current node value

• Returned node value = “best” child seen so far
  – Maximum child value seen so far for MAX nodes
  – Minimum child value seen so far for MIN nodes
  – If you prune, return to parent “best” child so far

• Returned node value is received by parent
Alpha-Beta Example Revisited

Do DF-search until first leaf

\[ \alpha = -\infty \]
\[ \beta = +\infty \]

\[ \alpha, \beta, \text{ initial values} \]

\[ \alpha = -\infty \]
\[ \beta = +\infty \]

\[ \alpha, \beta, \text{ passed to kids} \]

Review Detailed Example of Alpha-Beta Pruning in lecture slides.
Alpha-Beta Example (continued)

\[ \alpha = -\infty \]
\[ \beta = +\infty \]

MIN updates \( \beta \), based on kids
Alpha-Beta Example (continued)

MIN updates $\beta$, based on kids. No change.
Alpha-Beta Example (continued)

MAX updates $\alpha$, based on kids.

$\alpha = 3$

$\beta = +\infty$

3 is returned as node value.
Alpha-Beta Example (continued)

\[
\alpha = 3 \\
\beta = +\infty
\]

\[
\alpha, \beta, \text{ passed to kids} \\
\alpha = 3 \\
\beta = +\infty
\]
Alpha-Beta Example (continued)

\[ \alpha = 3 \]
\[ \beta = +\infty \]

MIN updates \( \beta \), based on kids.

\[ \alpha = 3 \]
\[ \beta = 2 \]
\(\alpha = 3\)
\(\beta = +\infty\)

\(\alpha \geq \beta\), so prune.
MAX updates $\alpha$, based on kids. No change.

$\alpha = 3$

$\beta = +\infty$

2 is returned as node value.

Review Detailed Example of Alpha-Beta Pruning in lecture slides.
Long Detailed Alpha-Beta Example

Branch nodes are labeled A-K for easy discussion

\( \alpha, \beta, \text{initial values} \rightarrow \alpha = -\infty \)
\( \beta = +\infty \)
Long Detailed Alpha-Beta Example

Note that search cut-off occurs at different depths

current $\alpha$, $\beta$, passed to kids

$\alpha = -\infty$
$\beta = +\infty$
kid = A

$\alpha = -\infty$
$\beta = +\infty$
kid = E
We also are running MiniMax search and recording node values within the triangles, without explicit comment.
Long Detailed Alpha-Beta Example

α = −∞
β = +∞

see next leaf,
MAX updates α

α = 5
β = +∞
kid = E

kid = A
Long Detailed Alpha-Beta Example

α = −∞
β = +∞

see next leaf,
MAX updates α

α = −∞
β = +∞
kid = A

α = 6
β = +∞
kid = E

α = 6
β = +∞
kid = E

α = 6
Long Detailed Alpha-Beta Example

\[ \alpha = -\infty \]
\[ \beta = +\infty \]

return node value, MIN updates \( \beta \)

\( \alpha = -\infty \)
\( \beta = 6 \)
kid=A

MAX

MIN
Long Detailed Alpha-Beta Example

current $\alpha$, $\beta$
passed to kid $F$

$\alpha = -\infty$
$\beta = +\infty$

$\alpha = -\infty$
$\beta = 6$
kid = A

$\alpha = -\infty$
$\beta = 6$
kid = F

Diagram of the alpha-beta pruning algorithm with values and decision points.

Nodes represent states in the game tree, with MAX and MIN nodes indicating the decision making for the maximizing and minimizing player respectively.

Key points:
- Initial $\alpha$ and $\beta$ values.
- Pruning occurs when $\alpha$ exceeds $\beta$.
- Children nodes provide the values for the parent.

The diagram illustrates how the algorithm efficiently prunes branches that cannot lead to a better outcome than already found.
Long Detailed Alpha-Beta Example

see first leaf, MAX updates $\alpha$

$\alpha = -\infty$
$\beta = +\infty$

$\alpha = 6$
$\beta = 6$
kid = A

$\alpha = 6$
kid = F

MAX

MIN
Long Detailed Alpha-Beta Example

\[ \alpha \geq \beta !! \]

Prune!!

\[ \alpha = -\infty \]
\[ \beta = +\infty \]

\[ \alpha = 6 \]
\[ \beta = 6 \]

kid = A

kid = F
return node value, MIN updates $\beta$, no change to $\beta$

If we had continued searching at node F, we would see the 9 from its third leaf. Our returned value would be 9 instead of 6. But at A, MIN would choose E(=6) instead of F(=9). Internal values may change; root values do not.
Long Detailed Alpha-Beta Example

see next leaf, MIN updates $\beta$, no change to $\beta$

$\alpha = -\infty$
$\beta = +\infty$

kid=A
return node value, $\alpha = 6$
MAX updates $\alpha$

$\beta = +\infty$
current $\alpha$, $\beta$, passed to kids

$\alpha = 6$
$\beta = +\infty$
kid = B

$\alpha = 6$
$\beta = +\infty$
kid = G
Long Detailed Alpha-Beta Example

see first leaf, MAX updates $\alpha$, no change to $\alpha$

$\alpha=6$
$\beta=+\infty$
kid=B

$\alpha=6$
$\beta=+\infty$
kid=G

MAX updates $\alpha$, no change to $\alpha$
Long Detailed Alpha-Beta Example

see next leaf, MAX updates $\alpha$, no change to $\alpha$

$\alpha=6$
$\beta=+\infty$
kid=B

$\alpha=6$
$\beta=+\infty$
kid=G

MAX updates $\alpha$, no change to $\alpha$
Long Detailed Alpha-Beta Example

\[ \alpha = 6 \]
\[ \beta = +\infty \]

Return node value, MIN updates \( \beta \)

\[ \alpha = 6 \]
\[ \beta = 5 \]

Kid = B
Note that we never find out what is the node value of H? But we have proven it doesn’t matter, so we don’t care.
Long Detailed Alpha-Beta Example

return node value, \( \alpha = 6 \)
MAX updates \( \alpha \),
no change to \( \alpha \)
Long Detailed Alpha-Beta Example

- Current $\alpha$, $\beta$, passed to $\text{kid}=C$
- $\alpha=6$
- $\beta=+\infty$
- $\text{kid}=C$

```
- A
  - 6
- B
  - 5
- C
  - ?
- D
- E
  - 6
- F
  - 6
- G
  - 5
- H
  - ?
- I
- J
- K
```
Long Detailed Alpha-Beta Example

see first leaf, MIN updates β

α=6
β =+∞

kid=C

α=6
β=9

MAX

MIN

MAX

MIN
Long Detailed Alpha-Beta Example

current $\alpha$, $\beta$, passed to kid I

$\alpha=6$
$\beta=+\infty$

$\alpha=6$
$\beta=9$
kid=C

$\alpha=6$
$\beta=9$
kid=I

current $\alpha$, $\beta$, passed to kid I
see first leaf, 
MAX updates $\alpha$, 
no change to $\alpha$

$\alpha=6$
$\beta=+\infty$

$\alpha=6$
$\beta=9$
kid=C

$\alpha=6$
$\beta=9$
kid=I

see first leaf,
MAX updates $\alpha$, 
no change to $\alpha$
see next leaf, MAX updates $\alpha$, no change to $\alpha$

$\alpha=6$
$\beta = +\infty$

$\alpha=6$
$\beta=9$
kid=C

$\alpha=6$
$\beta=9$
kid=I

see next leaf, MAX updates $\alpha$, no change to $\alpha$
Long Detailed Alpha-Beta Example

α=6
β =+∞

return node value, MIN updates β

α=6
β=6

kid=C

MAX

MIN

MAX

MAX

MAX
Long Detailed Alpha-Beta Example

\[ \alpha \geq \beta \text{ !!} \]

**Prune!!**

\[ \alpha = 6 \]
\[ \beta = +\infty \]

\[ \text{kid} = C \]

\[ \alpha = 6 \]
\[ \beta = 6 \]
return node value, \( \alpha = 6 \)

MAX updates \( \alpha \), no change to \( \alpha \)
Long Detailed Alpha-Beta Example

current $\alpha, \beta,$ passed to kid=D

$\alpha=6$
$\beta=+\infty$

$\alpha=6$
$\beta=+\infty$
kid=D
Long Detailed Alpha-Beta Example

See first leaf, MIN updates $\beta$

$\alpha = 6$
$\beta = +\infty$

Kid = D

MAX

MIN

MAX

MAX

MAX
Long Detailed Alpha-Beta Example

\[ \alpha \geq \beta!! \]

Prune!!

\[ \alpha = 6 \]

\[ \beta = +\infty \]

\[ \text{kid} = D \]

\[ \alpha = 6 \]

\[ \beta = 6 \]
return node value, $\alpha = 6$

MAX updates $\alpha$, $\beta = +\infty$

no change to $\alpha$
MAX moves to A, and expects to get 6

Although we may have changed some internal branch node return values, the final root action and expected outcome are identical to if we had not done alpha-beta pruning. Internal values may change; root values do not.
**Final Review**

<table>
<thead>
<tr>
<th>Week</th>
<th>Day Date</th>
<th>Quiz</th>
<th>Lecture 1 (1:00-2:20)</th>
<th>Lecture 2 (2:30-3:50)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Tue 5 Jul</td>
<td>Q2</td>
<td>Heuristic Search</td>
<td>Local Search</td>
</tr>
<tr>
<td></td>
<td>Thu 7 Jul</td>
<td></td>
<td><strong>Mid-term Review</strong></td>
<td><strong>Mid-term Exam</strong></td>
</tr>
<tr>
<td>4</td>
<td>Tue 12 Jul</td>
<td>Q3</td>
<td>Game Search A</td>
<td>Game Search B</td>
</tr>
<tr>
<td></td>
<td>Thu 14 Jul</td>
<td></td>
<td><strong>Constraint Satisfaction A</strong></td>
<td><strong>Constraint Satisfaction B</strong></td>
</tr>
<tr>
<td>5</td>
<td>Tue 19 Jul</td>
<td>Q4</td>
<td>Machine Learning A</td>
<td>Machine Learning B</td>
</tr>
<tr>
<td></td>
<td>Thu 21 Jul</td>
<td></td>
<td><strong>Final Review</strong></td>
<td><strong>Final Review</strong></td>
</tr>
<tr>
<td></td>
<td>Tue 26 Jul</td>
<td></td>
<td><strong>Final Exam</strong></td>
<td><strong>Final Exam</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week</th>
<th>Day Date</th>
<th></th>
<th>Lecture 1 (1:00-2:20)</th>
<th>Lecture 2 (2:30-3:50)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Tue 5 Jul</td>
<td></td>
<td>Chapter 3.5-3.7</td>
<td>Chapter 4.1-4.2</td>
</tr>
<tr>
<td></td>
<td>Thu 7 Jul</td>
<td></td>
<td><strong>Review all of the above</strong></td>
<td><strong>Mid-term Exam</strong></td>
</tr>
<tr>
<td>4</td>
<td>Tue 12 Jul</td>
<td></td>
<td>Chapter 5.1, 5.2, 5.4</td>
<td>Chapter 5.3 (optional: 5.5+)</td>
</tr>
<tr>
<td></td>
<td>Thu 14 Jul</td>
<td></td>
<td>Chapter 6.1-6.4, except 6.3.3</td>
<td><strong>same</strong></td>
</tr>
<tr>
<td>5</td>
<td>Tue 19 Jul</td>
<td></td>
<td>Chapter 18.1-18.4</td>
<td>Chapters 18.5-12, 20.1-2</td>
</tr>
<tr>
<td></td>
<td>Thu 21 Jul</td>
<td></td>
<td><strong>Review all of the above</strong></td>
<td><strong>Review all of the above</strong></td>
</tr>
<tr>
<td></td>
<td>Tue 26 Jul</td>
<td></td>
<td><strong>Final Exam</strong></td>
<td><strong>Final Exam</strong></td>
</tr>
</tbody>
</table>

- Please review your quizzes and old CS-171 tests
- At least one question from a prior quiz or old CS-171 test will appear on the Final Exam (and all other tests)
You Will Be Expected to Know

• Basic definitions (section 6.1)
  – What is a CSP?

• Backtracking search for CSPs (6.3)

• Variable ordering or selection (6.3.1)
  – Minimum Remaining Values (MRV) heuristic
  – Degree Heuristic (DH) (to unassigned variables)

• Value ordering or selection (6.3.1)
  – Least constraining value (LCV) heuristic
Constraint Satisfaction Problems

• What is a CSP?
  – Finite set of variables \( X_1, X_2, \ldots, X_n \)
  – Nonempty domain of possible values for each variable \( D_1, D_2, \ldots, D_n \)
  – Finite set of constraints \( C_1, C_2, \ldots, C_m \)
    • Each constraint \( C_i \) limits the values that variables can take,
      • e.g., \( X_1 \neq X_2 \)
    – Each constraint \( C_i \) is a pair \(<\text{scope}, \text{relation}>\)
      • Scope = Tuple of variables that participate in the constraint.
      • Relation = List of allowed combinations of variable values.
        May be an explicit list of allowed combinations.
        May be an abstract relation allowing membership testing and listing.

• CSP benefits
  – Standard representation pattern
  – Generic goal and successor functions
  – Generic heuristics (no domain specific expertise).
CSPs --- What is a solution?

• A *state* is an *assignment* of values to some or all variables.
  – An assignment is *complete* when every variable has an assigned value.
  – An assignment is *partial* when one or more variables have no assigned value.

• **Consistent assignment:**
  – An assignment that does not violate the constraints.

• A *solution* to a CSP is a *complete and consistent assignment*.
  – All variables are assigned, and none of the assignments violate the constraints.

• CSPs may require a solution that maximizes an *objective function*.
  – For simple linear cases, an optimal solution can be obtained by Linear Programming.

• Examples of Applications:
  – Scheduling the time of observations on the Hubble Space Telescope
  – Airline schedules
  – Cryptography
  – Computer vision, image interpretation
CSP example: map coloring

• Variables: WA, NT, Q, NSW, V, SA, T
• Domains: $D_i=\{\text{red, green, blue}\}$
• Constraints: adjacent regions must have different colors.
  • E.g. WA $\neq$ NT
CSP example: Map coloring solution

- A solution is:
  - A complete and consistent assignment.
  - All variables assigned, all constraints satisfied.

- E.g., \{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green\}
Constraint graphs

- Constraint graph:
  - nodes are variables
  - arcs are binary constraints

- Graph can be used to simplify search
  e.g. Tasmania is an independent subproblem
Backtracking search

• Similar to Depth-first search
  – At each level, picks a single variable to explore
  – Iterates over the domain values of that variable

• Generates kids one at a time, one per value

• Backtracks when a variable has no legal values left

• Uninformed algorithm
  – No good general performance
function BACKTRACKING-SEARCH(csp) return a solution or failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) return a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to CONSTRAINTS[csp] then
            add {var=value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var=value} from assignment
    return failure
function BACKTRACKING-SEARCH(csp) return a solution or failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) return a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to CONSTRAINTS[csp] then
            add {var=value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var=value} from assignment
    return failure
Minimum remaining values (MRV)

\[ \text{var} \leftarrow \text{SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp],assignment,csp)} \]

- A.k.a. most constrained variable heuristic
- **Heuristic Rule**: choose variable with the fewest legal moves
  - e.g., will immediately detect failure if X has no legal values
Degree heuristic for the initial variable

- **Heuristic Rule**: select variable that is involved in the largest number of constraints on other unassigned variables.

- Degree heuristic can be useful as a tie breaker.

- *In what order should a variable’s values be tried?*
function BACKTRACKING-SEARCH(csp) return a solution or failure
    return RECURSIVE-BACKTRACKING({}, csp)

define RECURSIVE-BACKTRACKING(assignment, csp) return a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to CONSTRAINTS[csp] then
            add {var=value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var=value} from assignment
        return failure
Least constraining value for value-ordering

- Least constraining value heuristic

- Heuristic Rule: given a variable choose the least constraining value
  - leaves the maximum flexibility for subsequent variable assignments
Minimum remaining values (MRV) vs. Least constraining value (LCV)

• Why do we want the MRV (minimum values, most constraining) for variable selection --- but the LCV (maximum values, least constraining) for value selection?

• Isn’t there a contradiction here?

• MRV for variable selection to reduces the branching factor.
  – Smaller branching factors lead to faster search.
  – Hopefully, when we get to variables with currently many values, constraint propagation (next lecture) will have removed some of their values and they’ll have small branching factors by then too.

• LCV for value selection increases the chance of early success.
  – If we are going to fail at this node, then we have to examine every value anyway, and their order makes no difference at all.
  – If we are going to succeed, then the earlier we succeed the sooner we can stop searching, so we want to succeed early.
  – LCV rules out the fewest possible solutions below this node, so we have the most chances for early success.
• Please review your quizzes and old CS-171 tests
  • At least one question from a prior quiz or old CS-171 test will appear on the Final Exam (and all other tests)
You Will Be Expected to Know

• Node consistency, arc consistency, path consistency, K-consistency (6.2)

• Forward checking (6.3.2)

• Local search for CSPs
  – Min-Conflict Heuristic (6.4)

• The structure of problems (6.5)
Forward checking

• Can we detect inevitable failure early?
  – And avoid it later?

• Forward checking idea: keep track of remaining legal values for unassigned variables.

• When a variable is assigned a value, update all neighbors in the constraint graph.
  • Forward checking stops after one step and does not go beyond immediate neighbors.

• Terminate search when any variable has no legal values.
Forward checking
Check only neighbors; delete any inconsistent values

- Assign \{WA=red\}

- Effects on other variables connected by constraints to WA
  - $NT$ can no longer be red
  - $SA$ can no longer be red
Forward checking

Check only neighbors; delete any inconsistent values

- Assign \( \{Q=\text{green}\} \)

- Effects on other variables connected by constraints with WA
  - \( NT \) can no longer be green
  - \( NSW \) can no longer be green
  - \( SA \) can no longer be green

- \textit{MRV heuristic} would automatically select NT or SA next

We already have failure, but Forward Checking is too simple to detect it now.
Forward checking

Check only neighbors; delete any inconsistent values

- If \( V \) is assigned \( blue \)

- Effects on other variables connected by constraints with WA
  - \( NSW \) can no longer be \( blue \)
  - \( SA \) is empty

- FC has detected that partial assignment is \( inconsistent \) with the constraints and backtracking can occur.
Constraint propagation

- Solving CSPs with combination of heuristics plus forward checking is more efficient than either approach alone.

- Forward Checking does not detect all failures when they become obvious.
  - E.g., NT and SA cannot both be blue in the example above, so failure.

- Higher-order Constraint Propagation can detect early failure.
  - However, to do so takes more computing time --- is it worth the extra effort??
Arc consistency algorithm (AC-3)

function AC-3(csp) returns false if inconsistency found, else true, may reduce csp domains
inputs: csp, a binary CSP with variables \{X_1, X_2, ..., X_n\}
local variables: queue, a queue of arcs, initially all the arcs in csp
/* initial queue must contain both (X_i, X_j) and (X_j, X_i) */
while queue is not empty do
  \( (X_i, X_j) \leftarrow \text{REMOVE-FIRST(queue)} \)
  if REMOVE-INCONSISTENT-VALUES(\( X_i, X_j \)) then
    if size of \( D_i = 0 \) then return false
    for each \( X_k \) in \text{NEIGHBORS}[X_i] \( - \) \{X_j\} do
      add \( (X_k, X_i) \) to queue if not already there
  return true

function REMOVE-INCONSISTENT-VALUES(\( X_i, X_j \)) returns true iff we delete a
value from the domain of \( X_i \)
removed \( \leftarrow \) false
for each \( x \) in \text{DOMAIN}[X_i] do
  if no value \( y \) in \text{DOMAIN}[X_j] allows \( (x,y) \) to satisfy the constraints
    between \( X_i \) and \( X_j \)
    then delete \( x \) from \text{DOMAIN}[X_i]; removed \( \leftarrow \) true
return removed

(from Mackworth, 1977)
Arc consistency (AC-3)
Like Forward Checking, but exhaustive until quiescence

- An Arc $X \rightarrow Y$ is consistent if for every value $x$ of $X$ there is some value $y$ of $Y$ consistent with $x$ (note that this is a directed property)

- Consider state of search after WA and Q are assigned & FC is done:

  $SA \rightarrow NSW$ is consistent if

  $SA=blue$ and $NSW=red$
Arc consistency (AC-3)
Like Forward Checking, but exhaustive until quiescence

• $X \rightarrow Y$ is consistent if
  for every value $x$ of $X$ there is some value $y$ of $Y$ consistent with $x$

• $NSW \rightarrow SA$ is consistent if
  $NSW=red$ and $SA=blue$
  $NSW=blue$ and $SA=???$

• **NSW=blue can be pruned**: No current domain value of $SA$ is consistent
Arc consistency (AC-3)
Like Forward Checking, but exhaustive until quiescence

- Enforce arc-consistency:
  Arc can be made consistent by removing *blue* from *NSW*

- Continue to propagate constraints....
  - Check $V \rightarrow NSW$
  - Not consistent for $V = red$
  - Remove red from $V$
Arc consistency (AC-3)
Like Forward Checking, but exhaustive until quiescence

Continue to propagate constraints....

• $SA \rightarrow NT$ is not consistent
  – and cannot be made consistent (Failure!)

• Arc consistency detects failure earlier than FC
  – Requires more computation: Is it worth the effort??
Arc consistency checking

- Can be run as a preprocessor, or after each assignment
  - As preprocessor before search: Removes obvious inconsistencies
  - After each assignment: Reduces search cost but increases step cost

- AC must be run repeatedly until no inconsistency remains
  - Like Forward Checking, but exhaustive until quiescence

- Trade-off
  - Requires overhead to do; but usually better than direct search
  - In effect, it can successfully eliminate large (and inconsistent) parts of the state space more effectively than can direct search alone

- Need a systematic method for arc-checking
  - If $X$ loses a value, neighbors of $X$ need to be rechecked:
    
    I.e., incoming arcs can become inconsistent again (outgoing arcs will stay consistent).
Local search for CSPs

- Use complete-state representation
  - Initial state = all variables assigned values
  - Successor states = change 1 (or more) values

- For CSPs
  - allow states with unsatisfied constraints (unlike backtracking)
  - operators \textbf{reassign} variable values
  - hill-climbing with n-queens is an example

- Variable selection: randomly select any conflicted variable

- Value selection: \textit{min-conflicts heuristic}
  - Select new value that results in a minimum number of conflicts with the other variables
Local search for CSP

**function** MIN-CONFLICTS(csp, max_steps) **return** solution or failure

**inputs:** csp, a constraint satisfaction problem

*max_steps*, the number of steps allowed before giving up

\[
\text{current} \leftarrow \text{an initial complete assignment for } \text{csp}
\]

\[
\text{for } i = 1 \text{ to } \text{max_steps} \text{ do}
\]

\[
\text{if } \text{current} \text{ is a solution for } \text{csp} \text{ then return } \text{current}
\]

\[
\text{var} \leftarrow \text{a randomly chosen, conflicted variable from VARIABLES[csp]}
\]

\[
\text{value} \leftarrow \text{the value } v \text{ for } \text{var} \text{ that minimizes CONFLICTS(var,v,current,csp)}
\]

\[
\text{set } \text{var} = \text{value} \text{ in current}
\]

**return** failure
Min-conflicts example 1

Use of min-conflicts heuristic in hill-climbing.
A two-step solution for an 8-queens problem using min-conflicts heuristic

At each stage a queen is chosen for reassignment in its column

The algorithm moves the queen to the min-conflict square breaking ties randomly.
Advantages of local search

- Local search can be particularly useful in an online setting
  - Airline schedule example
    - E.g., mechanical problems require than 1 plane is taken out of service
    - Can locally search for another “close” solution in state-space
    - Much better (and faster) in practice than finding an entirely new schedule

- The runtime of min-conflicts is roughly independent of problem size.
  - Can solve the millions-queen problem in roughly 50 steps.

- Why?
  - n-queens is easy for local search because of the relatively high density of solutions in state-space
Performance of min-conflicts

Given random initial state, can solve $n$-queens in almost constant time for arbitrary $n$ with high probability (e.g., $n = 10,000,000$)

The same appears to be true for any randomly-generated CSP except in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$

![Graph showing CPU time vs. R, with a critical ratio indicated.](image-url)
Hard satisfiability problems
Hard satisfiability problems

- Median runtime for 100 satisfiable random 3-CNF sentences, $n = 50$
Sudoku
Backtracking Search + Forward Checking

- $R = \frac{\text{number of initially filled cells}}{\text{total number of cells}}$
- Success Rate = $P(\text{random puzzle is solvable})$
- $\text{[total number of cells]} = 9 \times 9 = 81$
- $\text{[number of initially filled cells]} = \text{variable}$
### Final Review

<table>
<thead>
<tr>
<th>Week</th>
<th>Day Date</th>
<th>Quiz</th>
<th>Lecture 1 (1:00-2:20)</th>
<th>Lecture 2 (2:30-3:50)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Tue 5 Jul</td>
<td>Q2</td>
<td>Heuristic Search</td>
<td>Local Search</td>
</tr>
<tr>
<td></td>
<td>Thu 7 Jul</td>
<td></td>
<td><strong>Mid-term Review</strong></td>
<td><strong>Mid-term Exam</strong></td>
</tr>
<tr>
<td>4</td>
<td>Tue 12 Jul</td>
<td>Q3</td>
<td>Game Search A</td>
<td>Game Search B</td>
</tr>
<tr>
<td></td>
<td>Thu 14 Jul</td>
<td></td>
<td>Constraint Satisfaction A</td>
<td>Constraint Satisfaction B</td>
</tr>
<tr>
<td>5</td>
<td>Tue 19 Jul</td>
<td>Q4</td>
<td><strong>Machine Learning A</strong></td>
<td>Machine Learning B</td>
</tr>
<tr>
<td></td>
<td>Thu 21 Jul</td>
<td></td>
<td>Final Review</td>
<td>Final Review</td>
</tr>
<tr>
<td></td>
<td>Tue 26 Jul</td>
<td></td>
<td>Final Exam</td>
<td>Final Exam</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week</th>
<th>Day Date</th>
<th>Lecture 1 (1:00-2:20)</th>
<th>Lecture 2 (2:30-3:50)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Tue 5 Jul</td>
<td>Chapter 3.5-3.7</td>
<td>Chapter 4.1-4.2</td>
</tr>
<tr>
<td></td>
<td>Thu 7 Jul</td>
<td><strong>Review all of the above</strong></td>
<td><strong>Mid-term Exam</strong></td>
</tr>
<tr>
<td>4</td>
<td>Tue 12 Jul</td>
<td>Chapter 5.1, 5.2, 5.4</td>
<td>Chapter 5.3 (optional: 5.5+)</td>
</tr>
<tr>
<td></td>
<td>Thu 14 Jul</td>
<td>Chapter 6.1-6.4, except 6.3.3</td>
<td>same</td>
</tr>
<tr>
<td>5</td>
<td>Tue 19 Jul</td>
<td>Chapter 18.1-18.4</td>
<td>Chapters 18.5-12, 20.1-2</td>
</tr>
<tr>
<td></td>
<td>Thu 21 Jul</td>
<td><strong>Review all of the above</strong></td>
<td><strong>Review all of the above</strong></td>
</tr>
<tr>
<td></td>
<td>Tue 26 Jul</td>
<td>Final Exam (NO BREAK)</td>
<td>Final Exam</td>
</tr>
</tbody>
</table>

- Please review your quizzes and old CS-171 tests
- At least one question from a prior quiz or old CS-171 test will appear on the Final Exam (and all other tests)
You will be expected to know

- Understand Attributes, Error function, Classification, Hypothesis (Predictor function)

- What is Supervised Learning?

- Decision Tree Algorithm

- Entropy

- Information Gain

- Tradeoff between train and test with model complexity

- Cross validation
Terminology

• Attributes
  – Also known as features, variables, independent variables, covariates

• Target Variable
  – Also known as goal predicate, dependent variable, ...

• Classification
  – Also known as discrimination, supervised classification, ...

• Error function
  – Objective function, loss function, ...
Inductive learning

- Let \( x \) represent the input vector of attributes.
- Let \( f(x) \) represent the value of the target variable for \( x \)
  - The implicit mapping from \( x \) to \( f(x) \) is unknown to us.
  - We just have training data pairs, \( D = \{x, f(x)\} \) available.
- We want to learn a mapping from \( x \) to \( f \), i.e.,
  \[ h(x; \theta) \text{ is "close" to } f(x) \text{ for all training data points } x \]
  \( \theta \) are the parameters of our predictor \( h(\ldots) \).
- Examples:
  - \( h(x; \theta) = \text{sign}(w_1x_1 + w_2x_2 + w_3) \)
  - \( h_k(x) = (x_1 \text{ OR } x_2) \text{ AND } (x_3 \text{ OR NOT}(x_4)) \)
Empirical Error Functions

- Empirical error function:
  \[ E(h) = \sum_x \text{distance}[h(x; \theta), f] \]

  e.g., distance = squared error if \( h \) and \( f \) are real-valued (regression)
  distance = delta-function if \( h \) and \( f \) are categorical (classification)

  Sum is over all training pairs in the training data \( D \)

In learning, we get to choose

1. what class of functions \( h(\cdot) \) that we want to learn
   - potentially a huge space! ("hypothesis space")

2. what error function/distance to use
   - should be chosen to reflect real "loss" in problem
   - but often chosen for mathematical/algorithmsic convenience
Decision Tree Representations

- Decision trees are fully expressive
  - can represent any Boolean function
  - Every path in the tree could represent 1 row in the truth table
  - Yields an exponentially large tree
- Truth table is of size $2^d$, where $d$ is the number of attributes

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A xor B</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

A \[ \rightarrow \] B \[ \rightarrow \] A xor B
Decision Tree Representations

- Trees can be **very inefficient** for certain types of functions
  - Parity function: 1 only if an even number of 1’s in the input vector
    - Trees are very inefficient at representing such functions
  - Majority function: 1 if more than \( \frac{1}{2} \) the inputs are 1’s
    - Also inefficient
  - Simple DNF formulae can be represented easily
    - E.g., \( f = (A \text{ AND } B) \text{ OR } (\text{NOT}(A) \text{ AND } D) \)
    - DNF = disjunction of conjunctions

- **Decision trees are in effect DNF representations**
  - Often used in practice since they often result in compact approximate representations for complex functions
  - E.g., consider a truth table where most of the variables are irrelevant to the function
function DTL(examples, attributes, default) returns a decision tree

    if examples is empty then return default
    else if all examples have the same classification then return the classification
    else if attributes is empty then return MODE(examples)
    else
        best ← CHOOSE-ATTRIBUTE(attributes, examples)
        tree ← a new decision tree with root test best
        for each value $v_i$ of best do
            $\text{examples}_i ← \{\text{elements of examples with best } = v_i\}$
            $\text{subtree} ←$ DTL($\text{examples}_i$, attributes $-$ best, MODE(examples))
            add a branch to tree with label $v_i$ and subtree subtree
        return tree
Choosing an attribute

- Idea: a good attribute splits the examples into subsets that are (ideally) "all positive" or "all negative"

- *Patrons?* is a better choice
  - How can we quantify this?
  - One approach would be to use the classification error $E$ directly (greedily)
    - Empirically it is found that this works poorly
  - Much better is to use information gain (next slides)
Entropy

H(p) = entropy of distribution p = {p_i}
(called “information” in text)

= E \[p_i \log (1/p_i) \] = - p \log p - (1-p) \log (1-p)

Entropy is the expected amount of information we gain, given a probability distribution – its our average uncertainty.

In general, H(p) is
(a) maximized when all p_i are equal, and
(b) minimized (=0) when one of the p_i’s is 1 and all others zero.
Entropy with only 2 outcomes

Consider 2 class problem: \( p = \) probability of class 1, \( 1 - p = \) probability of class 2

In binary case, \( H(p) = - p \log p - (1-p) \log (1-p) \)

![Graph showing the entropy function for binary classification](image)
Information Gain

• $H(p) =$ entropy of class distribution at a particular node

• $H(p \mid A) =$ conditional entropy = average entropy of conditional class distribution, after we have partitioned the data according to the values in $A$

• $\text{Gain}(A) = H(p) - H(p \mid A)$

• Simple rule in decision tree learning
  - At each internal node, split on the node with the largest information gain (or equivalently, with smallest $H(p\mid A)$)

• Note that by definition, conditional entropy can’t be greater than the entropy
Overfitting and Underfitting
A Complex Model

$Y = \text{high-order polynomial in } X$
A Much Simpler Model

\[ Y = a \, X + b + \text{noise} \]
How Overfitting affects Prediction

Predictive Error

Underfitting

Overfitting

Error on Test Data

Error on Training Data

Ideal Range for Model Complexity

Model Complexity
Training and Validation Data

Idea: train each model on the "training data" and then test each model’s accuracy on the validation data.
The k-fold Cross-Validation Method

- Why just choose one particular 90/10 “split” of the data?
  - In principle we could do this multiple times

- “k-fold Cross-Validation” (e.g., k=10)
  - randomly partition our full data set into k disjoint subsets (each roughly of size n/k, n = total number of training data points)
    - for i = 1:10 (here k = 10)
      - train on 90% of data,
      - Acc(i) = accuracy on other 10%
    - end
  - Cross-Validation-Accuracy = \( \frac{1}{k} \sum_i \text{Acc}(i) \)
    - choose the method with the highest cross-validation accuracy
    - common values for k are 5 and 10
    - Can also do “leave-one-out” where k = n
Disjoint Validation Data Sets

- Full Data Set
- Training Data
- Validation Data (aka Test Data)

1st partition

2nd partition

Validation Data

3rd partition

4th partition

5th partition
Final Review

• Please review your quizzes and old CS-171 tests
  • At least one question from a prior quiz or old CS-171 test will appear on the Final Exam (and all other tests)
You will be expected to know

- Classifiers:
  - Decision trees
  - K-nearest neighbors
  - Naïve Bayes
  - Perceptrons, Support vector Machines (SVMs), Neural Networks
- Decision Boundaries for various classifiers
  - What can they represent conveniently? What not?
Classification in Euclidean Space

- A classifier is a partition of the space $\mathbf{x}$ into disjoint decision regions
  - Each region has a label attached
  - Regions with the same label need not be contiguous
  - For a new test point, find what decision region it is in, and predict the corresponding label

- Decision boundaries = boundaries between decision regions
  - The “dual representation” of decision regions

- We can characterize a classifier by the equations for its decision boundaries

- Learning a classifier $\Leftrightarrow$ searching for the decision boundaries that optimize our objective function
Decision Tree Example

Income

Debt

Note: tree boundaries are linear and axis-parallel
A Simple Classifier: Minimum Distance Classifier

- Training
  - Separate training vectors by class
  - Compute the mean for each class, \( \mu_k, \ k = 1, \ldots, m \)

- Prediction
  - Compute the closest mean to a test vector \( x' \) (using Euclidean distance)
  - Predict the corresponding class

- In the 2-class case, the decision boundary is defined by the locus of the hyperplane that is halfway between the 2 means and is orthogonal to the line connecting them

- This is a very simple-minded classifier – easy to think of cases where it will not work very well
Minimum Distance Classifier
Another Example: Nearest Neighbor Classifier

- The nearest-neighbor classifier
  - Given a test point $x'$, compute the distance between $x'$ and each input data point
  - Find the closest neighbor in the training data
  - Assign $x'$ the class label of this neighbor
  - (sort of generalizes minimum distance classifier to exemplars)

- If Euclidean distance is used as the distance measure (the most common choice), the nearest neighbor classifier results in piecewise linear decision boundaries

- Many extensions
  - e.g., kNN, vote based on k-nearest neighbors
  - $k$ can be chosen by cross-validation
Overall Boundary = Piecewise Linear

Feature 1

Feature 2

Decision Region for Class 1

Decision Region for Class 2
kNN Decision Boundary

- piecewise linear decision boundary
- Increasing $k$ "simplifies" decision boundary
  - Majority voting means less emphasis on individual points

$K = 1$  

$K = 3$
kNN Decision Boundary

- piecewise linear decision boundary
- Increasing $k$ "simplifies" decision boundary
  - Majority voting means less emphasis on individual points

$K = 5$  
$K = 7$
kNN Decision Boundary

- piecewise linear decision boundary
- Increasing $k$ “simplifies” decision boundary
  - Majority voting means less emphasis on individual points

K = 25

- True ("best") decision boundary
  - In this case is linear
  - Compared to kNN: not bad!
The kNN Classifier

- The kNN classifier often works very well.
- Easy to implement.
- Easy choice if characteristics of your problem are unknown.

- Can be sensitive to the choice of distance metric.
  - Often normalize feature axis values, e.g., z-score or [0, 1]
    - E.g., if one feature runs larger in magnitude than another
  - Categorical feature axes are difficult, e.g., Color as Red/Blue/Green
    - Maybe use the absolute differences of their wavelengths?
    - But what about French/Italian/Thai/Burger?
    - Often used: $\text{delta}(A,B) = \{\text{IF } (A=B) \text{ THEN 0 ELSE 1}\}$

- Can encounter problems with sparse training data.

- Can encounter problems in very high dimensional spaces.
  - Most points are corners.
  - Most points are at the edge of the space.
  - Most points are neighbors of most other points.
Linear Classifiers

- Linear classifier $\Leftrightarrow$ single linear decision boundary (for 2-class case)

- We can always represent a linear decision boundary by a linear equation:
  \[ w_1 x_1 + w_2 x_2 + \ldots + w_d x_d = \sum w_j x_j = w^t x = 0 \]

- In d dimensions, this defines a (d-1) dimensional hyperplane
  - d=3, we get a plane; d=2, we get a line

- For prediction we simply see if $\sum w_j x_j > 0$

- The $w_i$ are the weights (parameters)
  - Learning consists of searching in the d-dimensional weight space for the set of weights (the linear boundary) that minimizes an error measure
  - A threshold can be introduced by a “dummy” feature that is always one; it weight corresponds to (the negative of) the threshold

- Note that a minimum distance classifier is a special (restricted) case of a linear classifier
Minimum Error
Decision Boundary
The Perceptron Classifier  (pages 729-731 in text)

Input Attributes (Features)  Weights For Input Attributes  Bias or Threshold  Transfer Function  Output
The Perceptron Classifier  (pages 729-731 in text)

- The perceptron classifier is just another name for a linear classifier for 2-class data, i.e.,
  \[
  \text{output}(x) = \text{sign}( \sum w_j x_j )
  \]

- Loosely motivated by a simple model of how neurons fire

- For mathematical convenience, class labels are +1 for one class and -1 for the other

- Two major types of algorithms for training perceptrons
  - Objective function = classification accuracy ("error correcting")
  - Objective function = squared error (use gradient descent)
Two different types of perceptron output

x-axis below is \( f(x) = f = \) weighted sum of inputs
y-axis is the perceptron output

Thresholded output (step function), takes values +1 or -1

Sigmoid output, takes real values between -1 and +1

The sigmoid is in effect an approximation to the threshold function above, but has a gradient that we can use for learning

- Sigmoid function is defined as
  \[
  \sigma[f] = \left[ \frac{2}{1 + \exp[-f]} \right] - 1
  \]

- Derivative of sigmoid
  \[
  \frac{\partial \sigma}{\partial f}[f] = 0.5 \times (\sigma[f] + 1) \times (1 - \sigma[f])
  \]
Pseudo-code for Perceptron Training

```
Initialize each $w_j$ (e.g., randomly)

While (termination condition not satisfied)
  for $i = 1: N$  % loop over data points (an iteration)
    for $j = 1: d$  % loop over weights
      $\text{deltaw}_j = \eta \left( y(i) - \sigma[f(i)] \right) \partial\sigma[f(i)] x_j(i)$
      $w_j = w_j + \text{deltaw}_j$
    end
  calculate termination condition
end
```

- Inputs: $N$ features, $N$ targets (class labels), learning rate $\eta$
- Outputs: a set of learned weights
Support Vector Machines (SVM): “Modern perceptrons” (section 18.9, R&N)

- A modern linear separator classifier
  - Essentially, a perceptron with a few extra wrinkles

- Constructs a “maximum margin separator”
  - A linear decision boundary with the largest possible distance from the decision boundary to the example points it separates
  - “Margin” = Distance from decision boundary to closest example
  - The “maximum margin” helps SVMs to generalize well

- Can embed the data in a non-linear higher dimension space
  - Constructs a linear separating hyperplane in that space
    - This can be a non-linear boundary in the original space
      - Algorithmic advantages and simplicity of linear classifiers
      - Representational advantages of non-linear decision boundaries

- Currently most popular “off-the shelf” supervised classifier.
Constructs a “maximum margin separator”

**Figure 18.30**  FILES: . Support vector machine classification: (a) Two classes of points (black and white circles) and three candidate linear separators. (b) The maximum margin separator (heavy line), is at the midpoint of the margin (area between dashed lines). The support vectors (points with large circles) are the examples closest to the separator.
Can embed the data in a non-linear higher dimension space

Figure 18.31  FILES:  (a) A two-dimensional training set with positive examples as black circles and negative examples as white circles. The true decision boundary, $x_1^2 + x_2^2 \leq 1$, is also shown. (b) The same data after mapping into a three-dimensional input space ($x_1^2, x_2^2, \sqrt{2}x_1x_2$). The circular decision boundary in (a) becomes a linear decision boundary in three dimensions. Figure 18.29(b) gives a closeup of the separator in (b).
Multi-Layer Perceptrons (Artificial Neural Networks)  
(sections 18.7.3-18.7.4 in textbook)

• What if we took K perceptrons and trained them in parallel and then took a weighted sum of their sigmoidal outputs?  
  – This is a multi-layer neural network with a single “hidden” layer (the outputs of the first set of perceptrons)  
  – If we train them jointly in parallel, then intuitively different perceptrons could learn different parts of the solution  
    • They define different local decision boundaries in the input space

• What if we hooked them up into a general Directed Acyclic Graph?  
  – Can create simple “neural circuits” (but no feedback; not fully general)  
  – Often called neural networks with hidden units

• How would we train such a model?  
  – Backpropagation algorithm = clever way to do gradient descent  
  – Bad news: many local minima and many parameters  
    • training is hard and slow  
  – Good news: can learn general non-linear decision boundaries  
  – Generated much excitement in AI in the late 1980’s and 1990’s  
  – Techniques like boosting, support vector machines, are often preferred
Multi-Layer Perceptrons (Artificial Neural Networks)
(sections 18.7.3-18.7.4 in textbook)
Naïve Bayes Model

Basic Idea: We want to estimate \( P(C \mid X_1, \ldots, X_n) \), but it’s hard to think about computing the probability of a class from input attributes of an example.

Solution: Use Bayes’ Rule to turn \( P(C \mid X_1, \ldots, X_n) \) into an equivalent expression that involves only \( P(C) \) and \( P(X_i \mid C) \).

We can estimate \( P(C) \) easily from the frequency with which each class appears within our training data, and \( P(X_i \mid C) \) from the frequency with which each \( X_i \) appears in each class \( C \) within our training data.
Naïve Bayes Model

Bayes Rule: \[ P(C \mid X_1,\ldots X_n) \text{ is proportional to } P(C) \prod_i P(X_i \mid C) \]
[note: denominator \( P(X_1,\ldots X_n) \) is constant for all classes, may be ignored.]

Features \( X_i \) are conditionally independent given the class variable \( C \)
- choose the class value \( c_i \) with the highest \( P(c_i \mid x_1,\ldots, x_n) \)
- simple to implement, often works very well
- e.g., spam email classification: \( X \)'s = counts of words in emails

Conditional probabilities \( P(X_i \mid C) \) can easily be estimated from labeled data
- Problem: Need to avoid zeroes, e.g., from limited training data
- Solutions: Pseudo-counts, beta\([a,b]\) distribution, etc.
Naïve Bayes Model (2)

\[
P(C \mid X_1, \ldots X_n) = \alpha \prod P(X_i \mid C) P(C)
\]

Probabilities \(P(C)\) and \(P(X_i \mid C)\) can easily be estimated from labeled data

\[
P(C = c_j) \approx \#(\text{Examples with class label } c_j) / \#(\text{Examples})
\]

\[
P(X_i = x_{ik} \mid C = c_j) \\ \approx \#(\text{Examples with } X_i \text{ value } x_{ik} \text{ and class label } c_j) / \#(\text{Examples with class label } c_j)
\]

Usually easiest to work with logs

\[
\log [ P(C \mid X_1, \ldots X_n) ] \\ = \log \alpha + \Sigma [ \log P(X_i \mid C) + \log P(C) ]
\]

**DANGER:** Suppose ZERO examples with \(X_i\) value \(x_{ik}\) and class label \(c_j\) ?
An unseen example with \(X_i\) value \(x_{ik}\) will NEVER predict class label \(c_j\)!

Practical solutions: Pseudocounts, e.g., add 1 to every \#() , etc.
Theoretical solutions: Bayesian inference, beta distribution, etc.
Classifier Bias — Decision Tree or Linear Perceptron?

Figure 18.22  FILES:. Comparing the performance of perceptrons and decision trees. (a) Perceptrons are better at learning the majority function of 11 inputs. (b) Decision trees are better at learning the WillWait predicate in the restaurant example.
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
### Final Review

<table>
<thead>
<tr>
<th>Week</th>
<th>Day Date</th>
<th>Quiz</th>
<th>Lecture 1 (1:00-2:20)</th>
<th>Lecture 2 (2:30-3:50)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Tue 5 Jul</td>
<td>Q2</td>
<td>Heuristic Search</td>
<td>Local Search</td>
</tr>
<tr>
<td></td>
<td>Thu 7 Jul</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Tue 12 Jul</td>
<td>Q3</td>
<td>Game Search A</td>
<td>Game Search B</td>
</tr>
<tr>
<td></td>
<td>Thu 14 Jul</td>
<td></td>
<td>Constraint Satisfaction A</td>
<td>Constraint Satisfaction B</td>
</tr>
<tr>
<td>5</td>
<td>Tue 19 Jul</td>
<td>Q4</td>
<td>Machine Learning A</td>
<td>Machine Learning B</td>
</tr>
<tr>
<td></td>
<td>Thu 21 Jul</td>
<td></td>
<td>Final Review</td>
<td>Final Review</td>
</tr>
<tr>
<td></td>
<td>Tue 26 Jul</td>
<td></td>
<td>Final Exam</td>
<td>Final Exam</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week</th>
<th>Day Date</th>
<th>Lecture 1 (1:00-2:20)</th>
<th>Lecture 2 (2:30-3:50)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Tue 5 Jul</td>
<td>Chapter 3.5-3.7</td>
<td>Chapter 4.1-4.2</td>
</tr>
<tr>
<td></td>
<td>Thu 7 Jul</td>
<td>Review all of the above</td>
<td>Mid-term Exam</td>
</tr>
<tr>
<td>4</td>
<td>Tue 12 Jul</td>
<td>Chapter 5.1, 5.2, 5.4</td>
<td>Chapter 5.3 (optional: 5.5+)</td>
</tr>
<tr>
<td></td>
<td>Thu 14 Jul</td>
<td>Chapter 6.1-6.4, except 6.3.3</td>
<td>same</td>
</tr>
<tr>
<td>5</td>
<td>Tue 19 Jul</td>
<td>Chapter 18.1-18.4</td>
<td>Chapters 18.5-12, 20.1-2</td>
</tr>
<tr>
<td></td>
<td>Thu 21 Jul</td>
<td>Review all of the above</td>
<td>Review all of the above</td>
</tr>
<tr>
<td></td>
<td>Tue 26 Jul</td>
<td>Final Exam (NO BREAK)</td>
<td>Final Exam</td>
</tr>
</tbody>
</table>

- Please review your quizzes and old CS-171 tests
- At least one question from a prior quiz or old CS-171 test will appear on the Final Exam (and all other tests)