Mid-term Review
Chapters 2-7

• Review Agents (2.1-2.3)
• Review State Space Search
  • Problem Formulation (3.1, 3.3)
  • Blind (Uninformed) Search (3.4)
  • Heuristic Search (3.5)
  • Local Search (4.1, 4.2)
• Review Adversarial (Game) Search (5.1-5.4)
• Review Constraint Satisfaction (6.1-6.4)
• Review Propositional Logic (7.1-7.5)
• 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 mid-term (and all other tests)
Review Agents
Chapter 2.1-2.3

• Agent definition (2.1)

• Rational Agent definition (2.2)
  – Performance measure

• Task environment definition (2.3)
  – PEAS acronym
Agents

• An agent is anything that can be viewed as *perceiving* its *environment* through *sensors* and *acting* upon that environment through *actuators*

  **Human agent:**
  eyes, ears, and other organs for sensors;
  hands, legs, mouth, and other body parts for actuators

• **Robotic agent:**
  cameras and infrared range finders for sensors; various motors for actuators
Rational agents

- **Rational Agent**: For each possible percept sequence, a rational agent should select an action that is *expected* to maximize its **performance measure**, based on the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

- **Performance measure**: An objective criterion for success of an agent's behavior

- **E.g.**, performance measure of a vacuum-cleaner agent could be amount of dirt cleaned up, amount of time taken, amount of electricity consumed, amount of noise generated, etc.
Task Environment

• Before we design an intelligent agent, we must specify its “task environment”:

PEAS:

Performance measure
Environment
Actuators
Sensors
PEAS

- Example: Agent = Part-picking robot
- Performance measure: Percentage of parts in correct bins
- Environment: Conveyor belt with parts, bins
- Actuators: Jointed arm and hand
- Sensors: Camera, joint angle sensors
Review State Space Search
Chapters 3-4

• Problem Formulation (3.1, 3.3)
• Blind (Uninformed) Search (3.4)
  • Depth-First, Breadth-First, Iterative Deepening
  • Uniform-Cost, Bidirectional (if applicable)
  • Time? Space? Complete? Optimal?
• Heuristic Search (3.5)
  • A*, Greedy-Best-First
• Local Search (4.1, 4.2)
  • Hill-climbing, Simulated Annealing, Genetic Algorithms
  • Gradient descent
A problem is defined by five items:

- **initial state** e.g., "at Arad"
- **actions**
  - Actions(X) = set of actions available in State X
- **transition model**
  - Result(S,A) = state resulting from doing action A in state S
- **goal test**, e.g., x = "at Bucharest", Checkmate(x)
- **path cost** (additive, i.e., the sum of the step costs)
  - $c(x,a,y) =$ step cost of action $a$ in state $x$ to reach state $y$
  - assumed to be $\geq 0$

A solution is a sequence of actions leading from the initial state to a goal state
Vacuum world state space graph

- **states?** discrete: dirt and robot locations
- **initial state?** any
- **actions?** Left, Right, Suck
- **transition model?** as shown on graph
- **goal test?** no dirt at all locations
- **path cost?** 1 per action
Implementation: states vs. nodes

- A **state** is a (representation of) a physical configuration.
- A **node** is a data structure constituting part of a search tree.
- A node contains info such as:
  - state, parent node, action, path cost \(g(x)\), depth, etc.

The **Expand** function creates new nodes, filling in the various fields using the **Actions** \((S)\) and **Result** \((S, A)\) functions associated with the problem.
Tree search vs. Graph search
Review Fig. 3.7, p. 77

- Failure to detect repeated states can turn a linear problem into an exponential one!
- Test is often implemented as a hash table.
Solutions to Repeated States

• Graph search
  – never generate a state generated before
    • must keep track of all possible states (uses a lot of memory)
    • e.g., 8-puzzle problem, we have 9! = 362,880 states
    • approximation for DFS/DLS: only avoid states in its (limited) memory: avoid infinite loops by checking path back to root.
  – “visited?” test usually implemented as a hash table

State Space

Example of a Search Tree

faster, but memory inefficient
General tree search

function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(Initial-State[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node; ACTION[s] ← action; STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

closed ← an empty set
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    if STATE[node] is not in closed then
        add STATE[node] to closed
        fringe ← INSERTALL(EXPAND(node, problem), fringe)

Goal test after pop
Breadth-first graph search

function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure

node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0 if
problem.GOAL-TEST(node.STATE) then return SOLUTION(node) frontier ←
a FIFO queue with node as the only element
explored ← an empty set
loop do
  if EMPTY?(frontier) then return failure
  node ← POP(frontier) /* chooses the shallowest node in frontier */
  add node.STATE to explored
  for each action in problem.ACTIONS(node.STATE) do
    child ← CHILD-NODE(problem, node, action)
    if child.STATE is not in explored or frontier then
      if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
      frontier ← INSERT(child, frontier)

Figure 3.11 Breadth-first search on a graph.
Uniform cost search: sort by $g$

A* is identical but uses $f = g + h$

Greedy best-first is identical but uses $h$

**function** `UNIFORM-COST-SEARCH(problem)` **returns** a solution, or failure

node ← a node with `STATE = problem.INITIAL-STATE`, `PATH-COST = 0`

frontier ← a priority queue ordered by `PATH-COST`, with `node` as the only element

explored ← an empty set

loop do

  if `EMPTY?(frontier)` then return failure
  node ← `POP(frontier)` /* chooses the lowest-cost node in frontier */
  if `problem.GOAL-TEST(node.STATE)` then return `SOLUTION(node)`

  add `node.STATE` to `explored`

  for each action in `problem.ACTIONS(node.STATE)` do
    child ← `CHILD-NODE(problem, node, action)`
    if `child.STATE` is not in `explored` or `frontier` then
      frontier ← `INSERT(child, frontier)`
    else if `child.STATE` is in `frontier` with higher `PATH-COST` then
      replace that `frontier` node with `child`

**Figure 3.14** Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for `frontier` needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.
Depth-limited search & IDS

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

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

function Iterative-Deepening-Search( problem) returns a solution, or failure
  inputs: problem, a problem
  for depth ← 0 to ∞ do
    result ← Depth-Limited-Search(problem, depth)
    if result ≠ cutoff then return result
When to do Goal-Test? Summary

• For DFS, BFS, DLS, and IDS, the goal test is done when the child node is generated.
  – These are not optimal searches in the general case.
  – BFS and IDS are optimal if cost is a function of depth only; then, optimal goals are also shallowest goals and so will be found first.

• For GBFS the behavior is the same whether the goal test is done when the node is generated or when it is removed.
  – $h(\text{goal})=0$ so any goal will be at the front of the queue anyway.

• For UCS and A* the goal test is done when the node is removed from the queue.
  – This precaution avoids finding a short expensive path before a long cheap path.
Blind Search Strategies (3.4)

- Depth-first: Add successors to front of queue
- Breadth-first: Add successors to back of queue
- Uniform-cost: Sort queue by path cost $g(n)$
- Depth-limited: Depth-first, cut off at limit
- Iterated-deepening: Depth-limited, increasing
- Bidirectional: Breadth-first from goal, too.

**Review “Example hand-simulated search”**
- Slides 29-38, Lecture on “Uninformed Search”
A search strategy is defined by the order of node expansion.

Strategies are evaluated along the following dimensions:
- **completeness**: does it always find a solution if one exists?
- **time complexity**: number of nodes generated
- **space complexity**: maximum number of nodes in memory
- **optimality**: does it always find a least-cost solution?

Time and space complexity are measured in terms of:
- \(b\): maximum branching factor of the search tree
- \(d\): depth of the least-cost solution
- \(m\): maximum depth of the state space (may be \(\infty\))
- (for UCS: \(C^*\): true cost to optimal goal; \(\varepsilon > 0\): minimum step cost)
# Summary of algorithms

**Fig. 3.21, p. 91**

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening DLS</th>
<th>Bidirectional (if applicable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes[a]</td>
<td>Yes[a,b]</td>
<td>No</td>
<td>No</td>
<td>Yes[a]</td>
<td>Yes[a,d]</td>
</tr>
<tr>
<td>Time</td>
<td>O(b^d)</td>
<td>O(b^{1+C*/ε})</td>
<td>O(b^m)</td>
<td>O(b^l)</td>
<td>O(b^d)</td>
<td>O(b^{d/2})</td>
</tr>
<tr>
<td>Space</td>
<td>O(b^d)</td>
<td>O(b^{1+C*/ε})</td>
<td>O(bm)</td>
<td>O(bl)</td>
<td>O(bd)</td>
<td>O(b^{d/2})</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes[c]</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes[c]</td>
<td>Yes[c,d]</td>
</tr>
</tbody>
</table>

There are a number of footnotes, caveats, and assumptions.

See Fig. 3.21, p. 91.

[a] complete if b is finite
[b] complete if step costs $\geq \varepsilon > 0$
[c] optimal if step costs are all identical
   (also if path cost non-decreasing function of depth only)
[d] if both directions use breadth-first search
   (also if both directions use uniform-cost search with step costs $\geq \varepsilon > 0$)

Generally the preferred uninformed search strategy
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
Greedy best-first search

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

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

- Not an optimal search strategy
  - May perform well in practice
A* search

- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n) = $ cost so far to reach $n$
  - $h(n) = $ estimated cost from $n$ to goal
  - $f(n) = $ estimated total cost of path through $n$ to goal
- A* search sorts queue by $f(n)$
- Greedy Best First search sorts queue by $h(n)$
- Uniform Cost search sorts queue by $g(n)$
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

$$f(n') = g(n') + h(n') \quad \text{(by def.)}$$
$$= g(n) + c(n,a,n') + h(n') \quad (g(n') = g(n) + c(n,a,n'))$$
$$\geq g(n) + h(n) = f(n) \quad \text{(consistency)}$$
$$f(n') \geq f(n)$$

• 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
Local search algorithms (4.1, 4.2)

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

• State space = set of "complete" configurations
• Find configuration satisfying constraints, e.g., n-queens
• In such cases, we can use local search algorithms
• keep a single "current" state, try to improve it.
• Very memory efficient (only remember current state)
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;
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!!
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.

- **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.

Figure 4.4  FILES: figures/ridge.eps (Tue Nov 3 16:23:29 2009). Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.
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
```
Simulated annealing search

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

```python
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] - VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^{Δ E / T}
```
**P(accepting a worse successor)**

- Decreases as Temperature $T$ decreases
- Increases as $|ΔE|$ decreases
- (Sometimes step size also decreases with $T$)

| $|ΔE|$ | e$^{ΔE/T}$ | Temperature $T$ |
|-------|-------------|-----------------|
| High  | High        | Medium Low      |
| Low   | High        | Medium          |


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

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)

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 $\approx 20X$ 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 $\approx 20X$ 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>≈0.37</td>
<td>≈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
  - $P(\text{individual in next gen.}) = \frac{\text{individual fitness}}{\Sigma \text{population fitness}}$
- **Crossover** fit parents to yield next generation (off-spring)
- **Mutate** the offspring randomly with some low probability
Fitness function: #non-attacking queen pairs
- min = 0, max = 8 × 7/2 = 28
- $\Sigma_i$ fitness$_i = 24 + 23 + 20 + 11 = 78$
- $P(\text{child}_1 \text{ in next gen.}) = \text{fitness}_1/(\Sigma_i \text{fitness}_i) = 24/78 = 31\%$
- $P(\text{child}_2 \text{ in next gen.}) = \text{fitness}_2/(\Sigma_i \text{fitness}_i) = 23/78 = 29\%;$ etc

How to convert a fitness value into a probability of being in the next generation.
Gradient Descent

• Assume we have some cost-function: \( C(x_1,\ldots,x_n) \)
  and we want to minimize over continuous variables \( X_1, X_2, \ldots, X_n \)

1. Compute the gradient:
   \[
   \frac{\partial}{\partial x_i} C(x_1,\ldots,x_n) \quad \forall i
   \]

2. Take a small step downhill in the direction of the gradient:
   \[
   x_i \rightarrow x'_i = x_i - \lambda \frac{\partial}{\partial x_i} C(x_1,\ldots,x_n) \quad \forall i
   \]

3. Check if \( C(x_1,\ldots,x'_i,\ldots,x_n) < C(x_1,\ldots,x_i,\ldots,x_n) \)

4. If true then accept move, if not reject.

5. Repeat.
Review Adversarial (Game) Search
Chapter 5.1-5.4

- Minimax Search with Perfect Decisions (5.2)
  - Impractical in most cases, but theoretical basis for analysis
- Minimax Search with Cut-off (5.4)
  - Replace terminal leaf utility by heuristic evaluation function
- Alpha-Beta Pruning (5.3)
  - The fact of the adversary leads to an advantage in search!
- Practical Considerations (5.4)
  - Redundant path elimination, look-up tables, etc.
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.
  – (2nd ed.: Successor function: list of (move,state) pairs specifying legal moves.)
  – 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
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 \( a \in \text{ACTIONS}(state) \) MIN-VALUE(Result(state,a))

function MIN-VALUE(state) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
\( v \leftarrow +\infty \)
for \( a \) in ACTIONS(state) do
\( v \leftarrow \text{MAX}(v,\text{MIN-VALUE}(<\text{Result}(state,a)>)) \)
return \( v \)

function MAX-VALUE(state) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
\( v \leftarrow -\infty \)
for \( a \) in ACTIONS(state) do
\( v \leftarrow \text{MAX}(v,\text{MIN-VALUE}(<\text{Result}(state,a)>)) \)
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)
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 -infinity (loss) to +infinity (win) or [-1, +1].

• If the board evaluation is X for a player, it’s -X for the opponent
  – “Zero-sum game”
Evaluation functions

Black to move
White slightly better

White to move
Black winning

For chess, typically \textit{linear} weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with

$f_1(s) = (\text{number of white queens}) - (\text{number of black queens})$, etc.
Cutting off search

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

1. \textsc{Terminal}? is replaced by \textsc{Cutoff}?
2. \textsc{Utility} is replaced by \textsc{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 \text{human novice}
8-ply \approx \text{typical PC, human master}
12-ply \approx \text{Deep Blue, Kasparov}
General alpha-beta pruning

• Consider a node $n$ in the tree ---

• If player has a better choice at:
  – Parent node of $n$
  – Or any choice point further up

• Then $n$ will never be reached in play.

• Hence, when that much is known about $n$, it can be pruned.
Alpha-beta Algorithm

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

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

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

- 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 \( \alpha \geq \beta \)
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.
Alpha-Beta Example Revisited

Do DF-search until first leaf

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

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

\[ \alpha = -\infty \]
\[ \beta = +\infty \]
MIN updates $\beta$, based on kids.
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.

\[ \alpha = 3 \]
\[ \beta = 2 \]
\[ \alpha = 3 \quad \beta = +\infty \]

\[ \alpha \geq \beta, \text{ so prune.} \]
MAX updates $\alpha$, based on kids.
No change.

$\alpha = 3$
$\beta = +\infty$

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

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

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

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

MIN updates \(\beta\), based on kids.
\[\alpha = 3\]
\[\beta = 14\]
Alpha-Beta Example (continued)

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

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

\[ \alpha = 3 \]
\[ \beta = 5 \]
Alpha-Beta Example (continued)

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

2 is returned as node value.
Max calculates the same node value, and makes the same move!

Review Detailed Example of Alpha-Beta Pruning in lecture slides.
Review Constraint Satisfaction
Chapter 6.1-6.4

• What is a CSP

• Backtracking for CSP

• Local search for CSPs
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 $\langle$scope, relation$\rangle$
    • 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 a value.
  – An assignment is partial when some variables have no values.

• Consistent assignment
  – Assignment does not violate the constraints

• A solution to a CSP is a complete and consistent assignment.

• Some CSPs require a solution that maximizes an objective function.
CSP example: map coloring

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

- Solutions are assignments satisfying all constraints, 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
  (will return to graph structure later)
Backtracking example
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?*
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
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

- Assign \{WA=red\}

- Effects on other variables connected by constraints to WA
  - \textit{NT can no longer be red}
  - \textit{SA can no longer be red}
Forward checking

- 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

- MRV heuristic would automatically select NT or SA next
Arc consistency

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

- Put all arcs $X \rightarrow Y$ onto a queue ($X \rightarrow Y$ and $Y \rightarrow X$ both go on, separately)
- Pop one arc $X \rightarrow Y$ and remove any inconsistent values from $X$
- If any change in $X$, then put all arcs $Z \rightarrow X$ back on queue, where $Z$ is a neighbor of $X$
- Continue until queue is empty
Arc consistency

- \( X \rightarrow Y \) is consistent if for every value \( x \) of \( X \) there is some value \( y \) consistent with \( x \)

- \( NSW \rightarrow SA \) is consistent if
  - \( NSW=red \) and \( SA=blue \)
  - \( NSW=blue \) and \( SA=?？? \)
Arc consistency

- Can enforce arc-consistency:
  Arc can be made consistent by removing blue from NSW

- Continue to propagate constraints....
  - Check $V \rightarrow NSW$
  - Not consistent for $V = \text{red}$
  - Remove red from $V$
Arc consistency

- Continue to propagate constraints....

- $SA \rightarrow NT$ is not consistent
  - and cannot be made consistent

- Arc consistency detects failure earlier than FC
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 **reassign** variable values
  - hill-climbing with n-queens is an example

- Variable selection: randomly select any conflicted variable

- Value selection: *min-conflicts heuristic*
  - Select new value that results in a minimum number of conflicts with the other variables
Min-conflicts example 1

Use of min-conflicts heuristic in hill-climbing.
Review Propositional Logic
Chapter 7.1-7.5

- Definitions:
  - Syntax, Semantics, Sentences, Propositions, Entails, Follows, Derives, Inference, Sound, Complete, Model, Satisfiable, Valid (or Tautology)

- Syntactic Transformations:
  - E.g., \((A \Rightarrow B) \iff (\neg A \lor B)\)

- Semantic Transformations:
  - E.g., \((KB \models \alpha) \equiv (\models (KB \Rightarrow \alpha))\)

- Truth Tables:
  - Negation, Conjunction, Disjunction, Implication, Equivalence (Biconditional)

- Inference:
  - By Model Enumeration (truth tables)
  - By Resolution
Recap propositional logic: Syntax

- Propositional logic is the simplest logic – illustrates basic ideas

- The proposition symbols $P_1$, $P_2$ etc are sentences
  - If $S$ is a sentence, $\neg S$ is a sentence (negation)
  - If $S_1$ and $S_2$ are sentences, $S_1 \land S_2$ is a sentence (conjunction)
  - If $S_1$ and $S_2$ are sentences, $S_1 \lor S_2$ is a sentence (disjunction)
  - If $S_1$ and $S_2$ are sentences, $S_1 \Rightarrow S_2$ is a sentence (implication)
  - If $S_1$ and $S_2$ are sentences, $S_1 \Leftrightarrow S_2$ is a sentence (biconditional)
Recap propositional logic:

**Semantics**

Each model/world specifies true or false for each proposition symbol

E.g. \( P_{1,2}, P_{2,2}, P_{3,1} \)

- false  true  false

With these symbols, 8 possible models can be enumerated automatically.

Rules for evaluating truth with respect to a model \( m \):

- \( \neg S \) is true iff \( S \) is false
- \( S_1 \land S_2 \) is true iff \( S_1 \) is true and \( S_2 \) is true
- \( S_1 \lor S_2 \) is true iff \( S_1 \) is true or \( S_2 \) is true
- \( S_1 \Rightarrow S_2 \) is true iff \( S_1 \) is false or \( S_2 \) is true
  (i.e., \( S_1 \land \neg S_2 \) is false iff \( S_1 \) is true and \( S_2 \) is false)
- \( S_1 \leftrightarrow S_2 \) is true iff \( S_1 \Rightarrow S_2 \) is true and \( S_2 \Rightarrow S_1 \) is true

Simple recursive process evaluates an arbitrary sentence, e.g.,

\[ \neg P_{1,2} \land (P_{2,2} \lor P_{3,1}) = true \land (true \lor false) = true \land true = true \]
Recap propositional logic:

Truth tables for connectives

<table>
<thead>
<tr>
<th>$P$</th>
<th>$Q$</th>
<th>$\neg P$</th>
<th>$P \land Q$</th>
<th>$P \lor Q$</th>
<th>$P \Rightarrow Q$</th>
<th>$P \Leftrightarrow Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

**OR:** $P$ or $Q$ is true or both are true.

**XOR:** $P$ or $Q$ is true but not both.

**Implication is always true when the premises are False!**
Recap propositional logic:
Logical equivalence and rewrite rules

• To manipulate logical sentences we need some rewrite rules.
• Two sentences are logically equivalent iff they are true in same models: \( \alpha \equiv \beta \) iff \( \alpha \models \beta \) and \( \beta \models \alpha \)

\[
\begin{align*}
(\alpha \land \beta) & \equiv (\beta \land \alpha) \quad \text{commutativity of } \land \\
(\alpha \lor \beta) & \equiv (\beta \lor \alpha) \quad \text{commutativity of } \lor \\
((\alpha \land \beta) \land \gamma) & \equiv (\alpha \land (\beta \land \gamma)) \quad \text{associativity of } \land \\
((\alpha \lor \beta) \lor \gamma) & \equiv (\alpha \lor (\beta \lor \gamma)) \quad \text{associativity of } \lor \\
\neg(\neg \alpha) & \equiv \alpha \quad \text{double-negation elimination} \\
(\alpha \rightarrow \beta) & \equiv (\neg \beta \rightarrow \neg \alpha) \quad \text{contraposition} \\
(\alpha \rightarrow \beta) & \equiv (\neg \alpha \lor \beta) \quad \text{implication elimination} \\
(\alpha \leftrightarrow \beta) & \equiv ((\alpha \rightarrow \beta) \land (\beta \rightarrow \alpha)) \quad \text{biconditional elimination} \\
\neg(\alpha \land \beta) & \equiv (\neg \alpha \lor \neg \beta) \quad \text{de Morgan} \\
\neg(\alpha \lor \beta) & \equiv (\neg \alpha \land \neg \beta) \quad \text{de Morgan} \\
(\alpha \land (\beta \lor \gamma)) & \equiv ((\alpha \land \beta) \lor (\alpha \land \gamma)) \quad \text{distributivity of } \land \text{ over } \lor \\
(\alpha \lor (\beta \land \gamma)) & \equiv ((\alpha \lor \beta) \land (\alpha \lor \gamma)) \quad \text{distributivity of } \lor \text{ over } \land
\end{align*}
\]
Recap propositional logic: Entailment

- Entailment means that one thing follows from another:

\[ KB \models \alpha \]

- Knowledge base \( KB \) entails sentence \( \alpha \) if and only if \( \alpha \) is true in all worlds where \( KB \) is true

  - E.g., the KB containing “the Giants won and the Reds won” entails “The Giants won”.
  - E.g., \( x+y = 4 \) entails \( 4 = x+y \)
  - E.g., “Mary is Sue’s sister and Amy is Sue’s daughter” entails “Mary is Amy’s aunt.”
Review: Models (and in FOL, Interpretations)

- **Models** are formal worlds in which truth can be evaluated.

- We say *m* is a model of a sentence *α* if *α* is true in *m*.

- *M(α)* is the set of all models of *α*.

- Then *KB |= α* iff *M(KB) ⊆ M(α)*.
  - E.g. *KB* = “Mary is Sue’s sister and Amy is Sue’s daughter.”
  - *α* = “Mary is Amy’s aunt.”

- Think of *KB* and *α* as constraints, and of models *m* as possible states.

- *M(KB)* are the solutions to *KB* and *M(α)* the solutions to *α*.

- Then, *KB |= α*, i.e., *|= (KB ⇒ α)*, when all solutions to *KB* are also solutions to *α*. 
Review: Wumpus models

- $KB =$ all possible wumpus-worlds consistent with the observations and the “physics” of the Wumpus world.
Review: Wumpus models

\[ \alpha_1 = \text{"[1,2] is safe"}, \ KB \models \alpha_1, \text{proved by model checking.} \]

Every model that makes KB true also makes \( \alpha_1 \) true.
$\alpha_2 = "[2,2] is safe", KB \models \alpha_2$
If KB is true in the real world, then any sentence $\alpha$ entailed by KB and any sentence $\alpha$ derived from KB by a sound inference procedure is also true in the real world.
Schematic Example: Follows, Entails, and Derives

**Inference**

“Mary is Sue’s sister and Amy is Sue’s daughter.”

“An aunt is a sister of a parent.”

**Derives**

Derives

Is it provable?

“Mary is Amy’s aunt.”

**Entails**

Entails

Is it true?

“Mary is Amy’s aunt.”

**Follows**

Follows

Is it the case?

“Mary is Amy’s aunt.”

**World**

Mary

Sister

Sue

Daughter

Amy

Mary

Aunt

Amy
Recap propositional logic: Validity and satisfiability

A sentence is **valid** if it is true in **all** models,
e.g., True, A ∨ ¬A, A ⇒ A, (A ∧ (A ⇒ B)) ⇒ B

Validity is connected to inference via the **Deduction Theorem**:
\[ KB \vdash \alpha \text{ if and only if } (KB \Rightarrow \alpha) \text{ is valid} \]

A sentence is **satisfiable** if it is true in **some** model
e.g., A ∨ B, C

A sentence is **unsatisfiable** if it is false in **all** models
e.g., A ∧ ¬A

Satisfiability is connected to inference via the following:
\[ KB \not\vdash A \text{ if and only if } (KB \land \neg A) \text{ is unsatisfiable} \]
(there is no model for which KB is true and A is false)
Inference Procedures

- $KB \vdash_i A$ means that sentence $A$ can be derived from $KB$ by procedure $i$

- **Soundness**: $i$ is sound if whenever $KB \vdash_i \alpha$, it is also true that $KB \models \alpha$
  - *(no wrong inferences, but maybe not all inferences)*

- **Completeness**: $i$ is complete if whenever $KB \models \alpha$, it is also true that $KB \vdash_i \alpha$
  - *(all inferences can be made, but maybe some wrong extra ones as well)*

- Entailment can be used for inference (Model checking)
  - enumerate all possible models and check whether $\alpha$ is true.
  - For $n$ symbols, time complexity is $O(2^n)$...

- Inference can be done directly on the sentences
  - Forward chaining, backward chaining, resolution (see FOPC, later)
Resolution = Efficient Implication

Recall that \((A \Rightarrow B) = (\neg A \lor B)\)
and so:

\[
\begin{align*}
(Y \lor X) &= (\neg X \Rightarrow Y) \\
(\neg Y \lor Z) &= (Y \Rightarrow Z)
\end{align*}
\]

which yields:

\[
(Y \lor X) \land (\neg Y \lor Z) = (\neg X \Rightarrow Z) = (X \lor Z)
\]

Recall: All clauses in KB are conjoined by an implicit AND (= CNF representation).
Resolution Examples

- Resolution: inference rule for CNF: sound and complete! *

\[(A \lor B \lor C)\]
\[(-A)\]
\[\therefore (B \lor C)\]

“If A or B or C is true, but not A, then B or C must be true.”

\[(A \lor B \lor C)\]
\[(-A \lor D \lor E)\]
\[\therefore (B \lor C \lor D \lor E)\]

“If A is false then B or C must be true, or if A is true then D or E must be true, hence since A is either true or false, B or C or D or E must be true.”

\[(A \lor B)\]
\[(-A \lor B)\]
\[\therefore (B \lor B) \equiv B\]

Simplification is done always.

* Resolution is “refutation complete” in that it can prove the truth of any entailed sentence by refutation.
* You can start two resolution proofs in parallel, one for the sentence and one for its negation, and see which branch returns a correct proof.
Only Resolve **ONE** Literal Pair!
If more than one pair, result always = TRUE.

**Useless!!** Always simplifies to TRUE!!

No!  
(OR \(A\) \(B\) \(C\) \(D\))  
(OR \(\neg A\) \(\neg B\) \(F\) \(G\))  
___________________________  
(OR \(C\) \(D\) \(F\) \(G\))  
**No!**

Yes! (but = TRUE)  
(OR \(A\) \(B\) \(C\) \(D\))  
(OR \(\neg A\) \(\neg B\) \(F\) \(G\))  
___________________________  
(OR \(B\) \(\neg B\) \(C\) \(D\) \(F\) \(G\))  
**Yes! (but = TRUE)**

No!  
(OR \(A\) \(B\) \(C\) \(D\))  
(OR \(\neg A\) \(\neg B\) \(\neg C\))  
___________________________  
(OR \(D\))  
**No!**

Yes! (but = TRUE)  
(OR \(A\) \(B\) \(C\) \(D\))  
(OR \(\neg A\) \(\neg B\) \(\neg C\))  
___________________________  
(OR \(A\) \(\neg A\) \(B\) \(\neg B\) \(D\))  
**Yes! (but = TRUE)**
Resolution Algorithm

- The resolution algorithm tries to prove: $KB \models \alpha$ equivalent to $KB \land \neg \alpha$ unsatisfiable.

- Generate all new sentences from KB and the (negated) query.
- One of two things can happen:

1. We find $P \land \neg P$ which is unsatisfiable. I.e. we can entail the query.

2. We find no contradiction: there is a model that satisfies the sentence $KB \land \neg \alpha$ (non-trivial) and hence we cannot entail the query.
Resolution example

- \( KB = (B_{1,1} \iff (P_{1,2} \lor P_{2,1})) \land \neg B_{1,1} \)
- \( \alpha = \neg P_{1,2} \)

\[ KB \land \neg \alpha \]

True!

False in all worlds
**Detailed Resolution Proof Example**

- In words: *If the unicorn is mythical, then it is immortal, but if it is not mythical, then it is a mortal mammal. If the unicorn is either immortal or a mammal, then it is horned. The unicorn is magical if it is horned.*

  *Prove that the unicorn is both magical and horned.*

\[
\begin{align*}
( \neg Y \land \neg R ) & \quad (M \land Y) & \quad (R \land Y) & \quad (H \land \neg M) \\
(H \land R) & \quad (\neg H \land G) & \quad (\neg G \land \neg H) & \\
\end{align*}
\]

- **Fourth, produce a resolution proof ending in ( ):**
  - Resolve \((\neg H \land \neg G)\) and \((\neg H \land G)\) to give \((\neg H)\)
  - Resolve \((\neg Y \land \neg R)\) and \((Y \land M)\) to give \((\neg R \land M)\)
  - Resolve \((\neg R \land M)\) and \((R \land H)\) to give \((M \land H)\)
  - Resolve \((M \land H)\) and \((\neg M \land H)\) to give \((H)\)
  - Resolve \((\neg H)\) and \((H)\) to give ( )

- Of course, there are many other proofs, which are OK iff correct.
Propositional Logic --- Summary

- Logical agents apply inference to a knowledge base to derive new information and make decisions

- Basic concepts of logic:
  - syntax: formal structure of sentences
  - semantics: truth of sentences wrt models
  - entailment: necessary truth of one sentence given another
  - inference: deriving sentences from other sentences
  - soundness: derivations produce only entailed sentences
  - completeness: derivations can produce all entailed sentences
  - valid: sentence is true in every model (a tautology)

- Logical equivalences allow syntactic manipulations

- Propositional logic lacks expressive power
  - Can only state specific facts about the world.
  - Cannot express general rules about the world
    (use First Order Predicate Logic instead)
Mid-term Review
Chapters 2-7

• Review Agents (2.1-2.3)
• Review State Space Search
  • Problem Formulation (3.1, 3.3)
  • Blind (Uninformed) Search (3.4)
  • Heuristic Search (3.5)
  • Local Search (4.1, 4.2)
• Review Adversarial (Game) Search (5.1-5.4)
• Review Constraint Satisfaction (6.1-6.4)
• Review Propositional Logic (7.1-7.5)
• 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 mid-term (and all other tests)