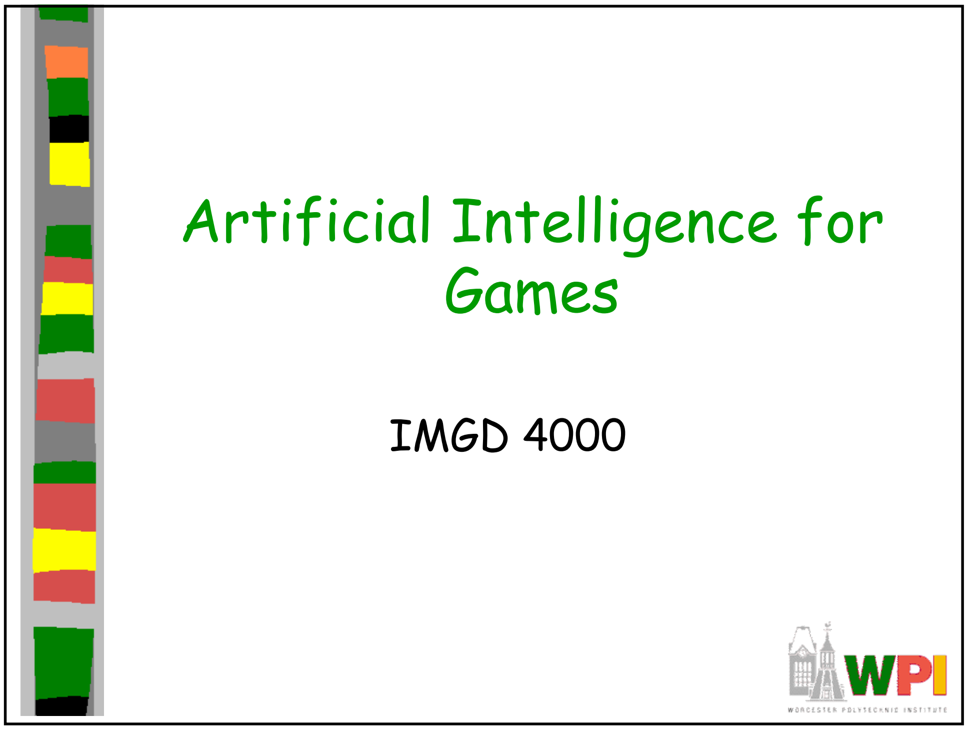
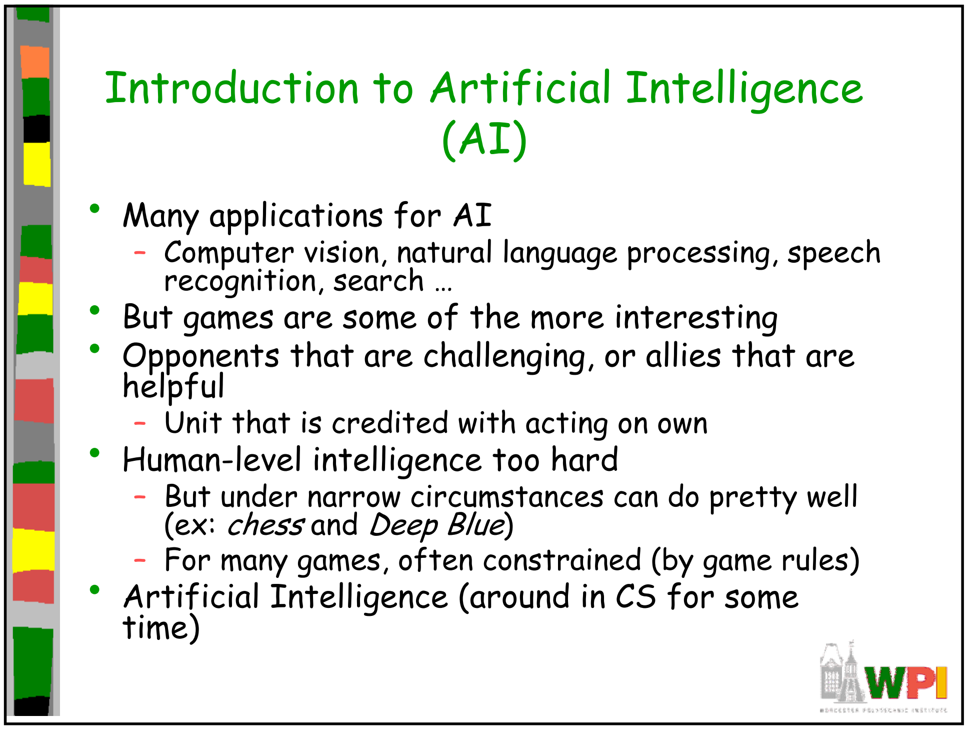


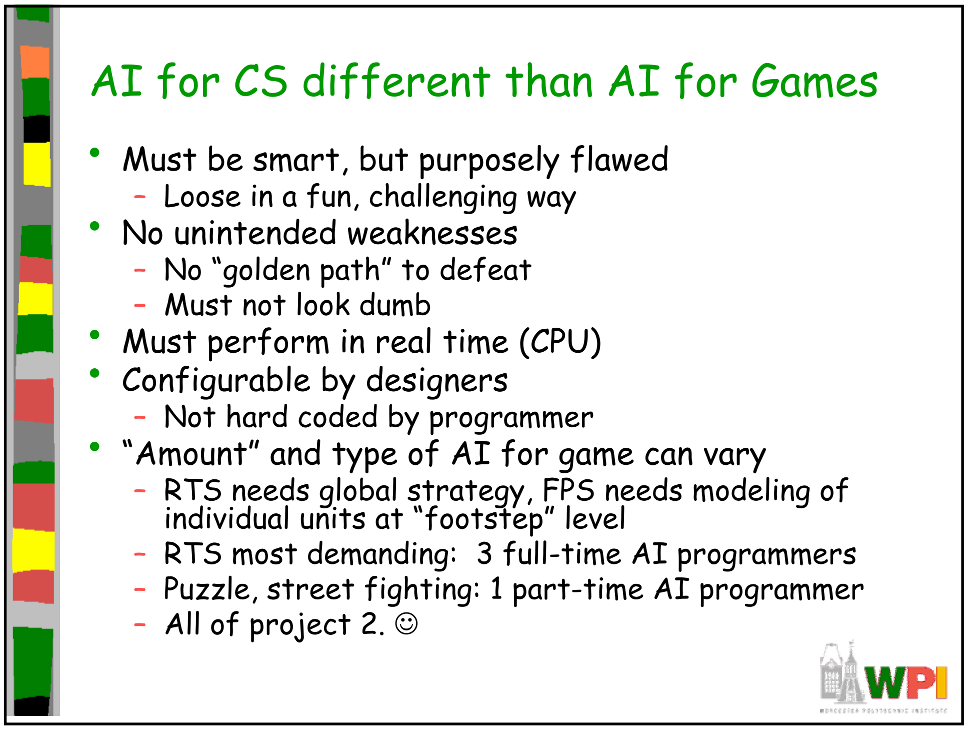
Artificial Intelligence for Games

IMGD 4000



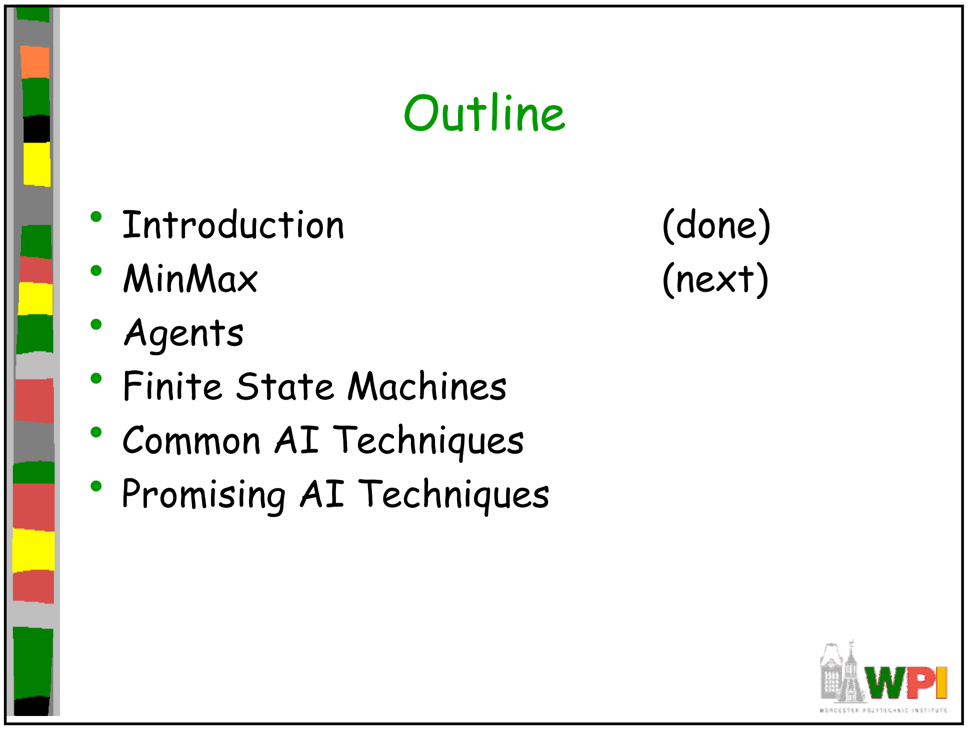
Introduction to Artificial Intelligence (AI)

- Many applications for AI
  - Computer vision, natural language processing, speech recognition, search …
- But games are some of the more interesting
- Opponents that are challenging, or allies that are helpful
  - Unit that is credited with acting on own
- Human-level intelligence too hard
  - But under narrow circumstances can do pretty well (ex: *chess* and *Deep Blue*)
  - For many games, often constrained (by game rules)
- Artificial Intelligence (around in CS for some time)



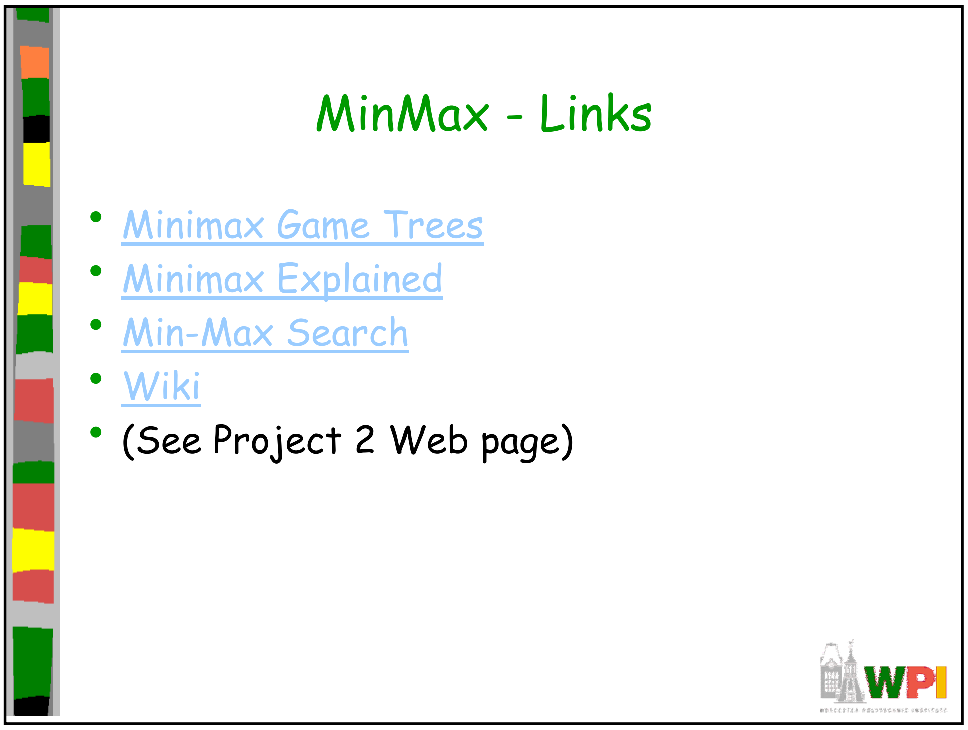
AI for CS different than AI for Games

- Must be smart, but purposely flawed
  - Loose in a fun, challenging way
- No unintended weaknesses
  - No “golden path” to defeat
  - Must not look dumb
- Must perform in real time (CPU)
- Configurable by designers
  - Not hard coded by programmer
- “Amount” and type of AI for game can vary
  - RTS needs global strategy, FPS needs modeling of individual units at “footstep” level
  - RTS most demanding: 3 full-time AI programmers
  - Puzzle, street fighting: 1 part-time AI programmer
  - All of project 2. ☺



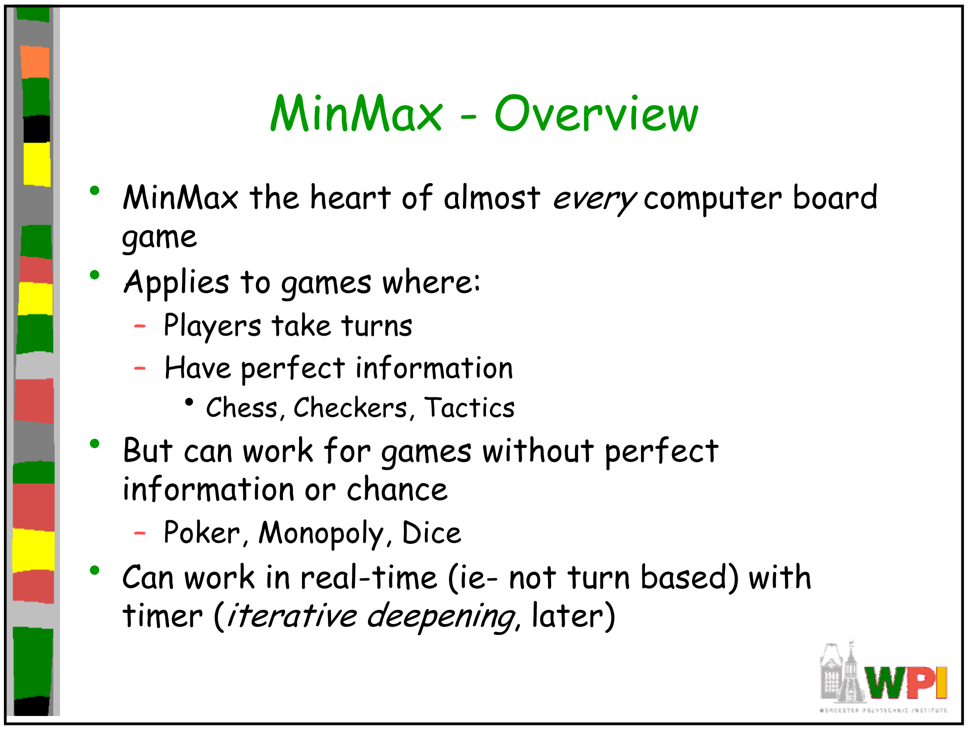
Outline

- Introduction (done)
- MinMax (next)
- Agents
- Finite State Machines
- Common AI Techniques
- Promising AI Techniques



MinMax - Links

- Minimax Game Trees
- Minimax Explained
- Min-Max Search
- Wiki
- (See Project 2 Web page)

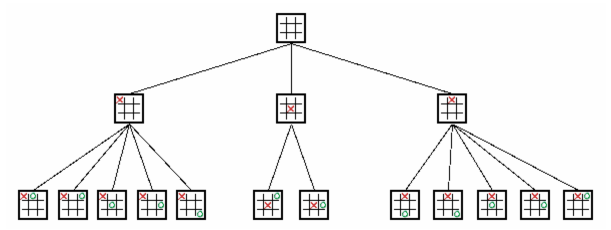


MinMax - Overview

- MinMax the heart of almost *every* computer board game
- Applies to games where:
  - Players take turns
  - Have perfect information
    - Chess, Checkers, Tactics
- But can work for games without perfect information or chance
  - Poker, Monopoly, Dice
- Can work in real-time (ie- not turn based) with timer (*iterative deepening*, later)

MinMax - Overview

- Search tree
  - *Squares* represent decision states (ie- after a move)
  - *Branches* are decisions (ie- the move)
  - Start at root
  - Nodes at end are leaf nodes
  - Ex: Tic-Tac-Toe (symmetrical positions removed)



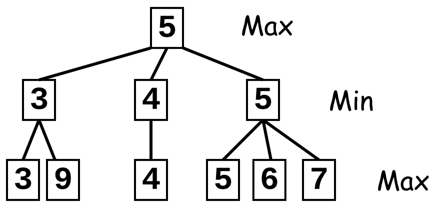
- Unlike binary trees can have any number of children
  - Depends on the game situation
- Levels usually called *plies* (a *ply* is one level)
  - Each ply is where "turn" switches to other player
- Players called *Min* and *Max* (next)

MaxMin - Algorithm

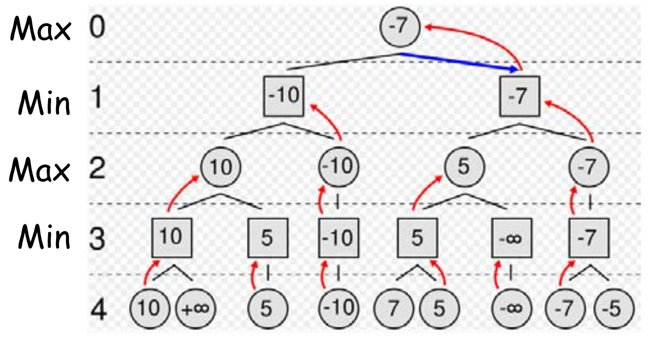
- Named *MinMax* because of algorithm behind data structure
- Assign points to the outcome of a game
  - Ex: Tic-Tac-Toe: *X* wins, value of 1. *O* wins, value -1.
- Max (X) tries to maximize point value, while Min (O) tries to minimize point value
- Assume both players play to best of their ability
  - Always make a move to minimize or maximize points
- So, in choosing, Max will choose best move to get highest points, assuming Min will choose best move to get lowest points

MinMax – First Example

- Max's turn
- Would like the "9" points (the maximum)
- But if choose left branch, Min will choose move to get 3
  - → left branch has a value of 3
- If choose right, Min can choose any one of 5, 6 or 7 (will choose 5, the minimum)
  - → right branch has a value of 5
- Right branch is largest (the maximum) so choose that move



MinMax – Second Example



- Max's turn
- Circles represent Max, Squares represent Min
- Values inside represent the value the MinMax algorithm
- Red arrows represent the chosen move
- Numbers on left represent tree depth
- Blue arrow is the chosen move

MinMax and Chess

- With full tree, can determine best possible move
- However, full tree impossible for some games! Ex: Chess
  - At a given time, chess has ~ 35 legal moves. Exponential growth:
    - 35 at one ply, 35^2 = 1225 at two plies … 35^6 = 2 billion and 35^{10} = 2 quadrillion
  - Games can last 40 moves (or more), so 35^{40} … Stars in universe: ~ 2^{28}
- For large games (Chess) can't see end of the game. Must *estimate* winning or losing from top portion
  - `Evaluate()` function to guess end given board
  - A numeric value, much smaller than victory (ie- Checkmate for Max will be one million, for Min minus one million)
- So, computer's strength at chess comes from:
  - How deep can search
  - How well can evaluate a board position
  - (In some sense, like a human – a chess grand master can evaluate board better and can look further ahead)

MinMax – Pseudo Code (1 of 3)

```
int MinMax(int depth) {
  // White is Max, Black is Min
  if (turn == WHITE)
     return Max(depth);
  else
     return Min(depth);
}
```

- Then, call with:

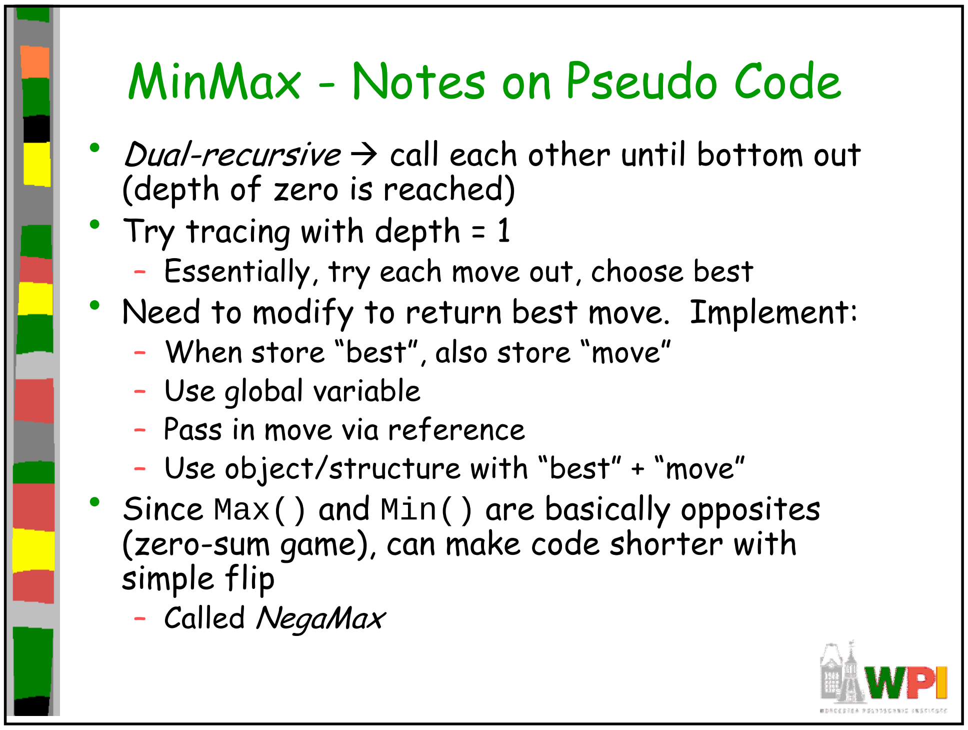
```
value = MinMax(5); // search 5 plies
```

MinMax - Pseudo Code (2 of 3)

```
int Max(int depth) {
  int best = -INFINITY;  // first move is best
  if (depth == 0)
      return Evaluate();
  GenerateLegalMoves();
  while (MovesLeft()) {
      MakeNextMove();
      val = Min(depth – 1);  // Min’s turn next
      UnMakeMove();
      if (val > best)
            best = val;
  }
  return best;
}
```

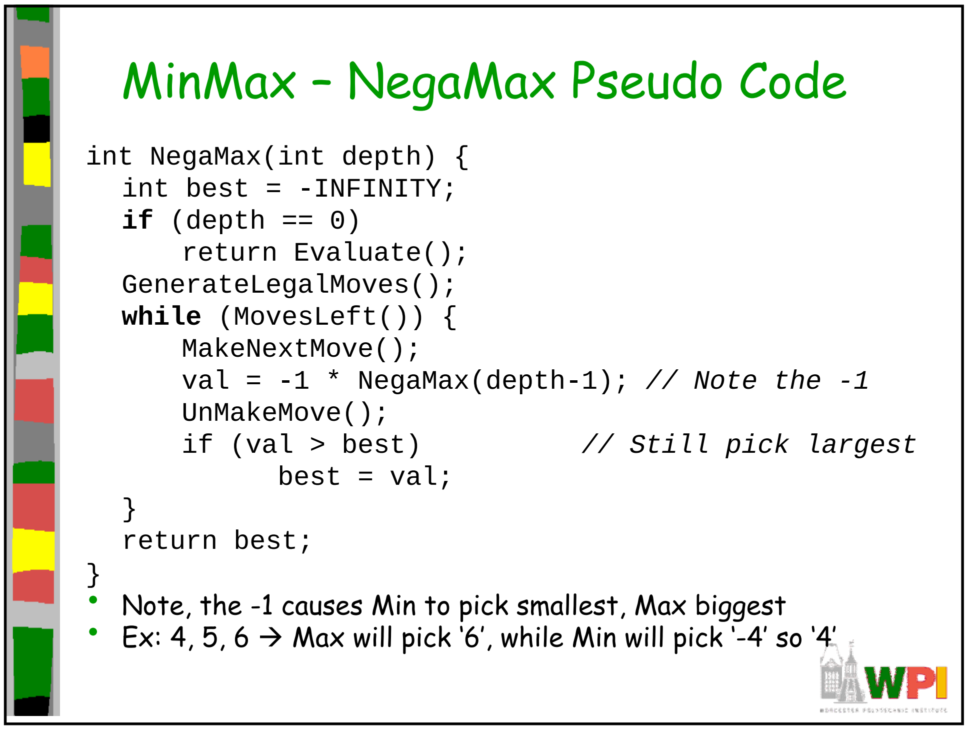
MinMax - Pseudo Code (3 of 3)

```
int Min(int depth) {
  int best = INFINITY; // ← different than MAX
  if (depth == 0)
      return Evaluate();
  GenerateLegalMoves();
  while (MovesLeft()) {
      MakeNextMove();
      val = Max(depth – 1); // Max’s turn next
      UnMakeMove();
      if (val < best) // ← different than MAX
            best = val;
  }
  return best;
}
```



MinMax - Notes on Pseudo Code

- *Dual-recursive* → call each other until bottom out (depth of zero is reached)
- Try tracing with depth = 1
  - Essentially, try each move out, choose best
- Need to modify to return best move. Implement:
  - When store "best", also store "move"
  - Use global variable
  - Pass in move via reference
  - Use object/structure with "best" + "move"
- Since `Max()` and `Min()` are basically opposites (zero-sum game), can make code shorter with simple flip
  - Called *NegaMax*



MinMax – NegaMax Pseudo Code

```
int NegaMax(int depth) {
   int best = -INFINITY;
   if (depth == 0)
        return Evaluate();
   GenerateLegalMoves();
   while (MovesLeft()) {
        MakeNextMove();
        val = -1 * NegaMax(depth-1); // Note the -1
        UnMakeMove();
        if (val > best)          // Still pick largest
                best = val;
   }
   return best;
}
```

- Note, the -1 causes Min to pick smallest, Max biggest
- Ex: 4, 5, 6 → Max will pick '6', while Min will pick '-4' so '4'

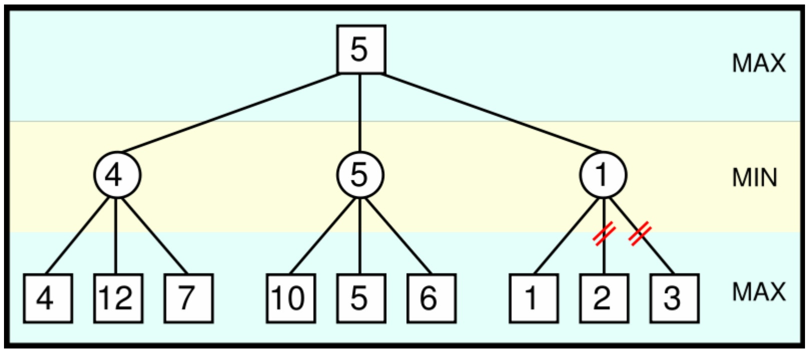
MinMax – AlphaBeta Pruning

- MinMax searches entire tree, even if in some cases the rest can be ignored
- Example – Enemy lost bet. Owes you one thing from bag. You choose bag, but he chooses thing. Go through bags one item at a time.
  - First bag: Sox tickets, sandwich, $20
    - He'll choose sandwich
  - Second bag: Dead fish, …
    - He'll choose fish. Doesn't matter if rest is car, $500, Yankee's tickets … Don't need to look further. Can prune.
- In general, stop evaluating move when find worse than previously examined move

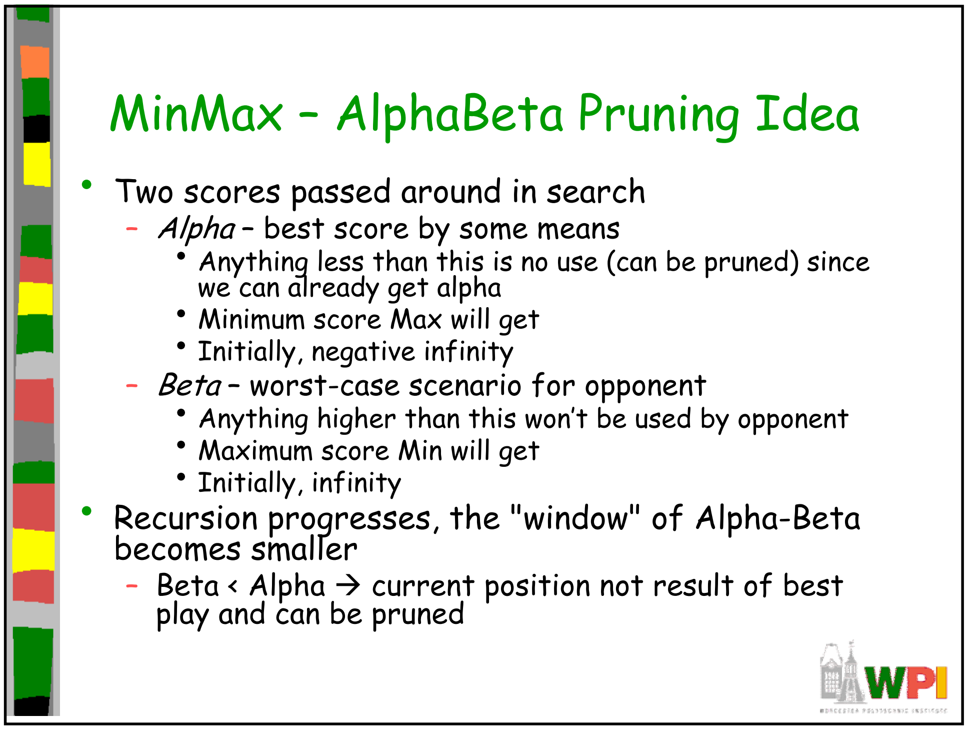
  → Does not benefit the player to play that move, it need not be evaluated any further.

  → Save processing time *without affecting final result*

MinMax – AlphaBeta Pruning Example

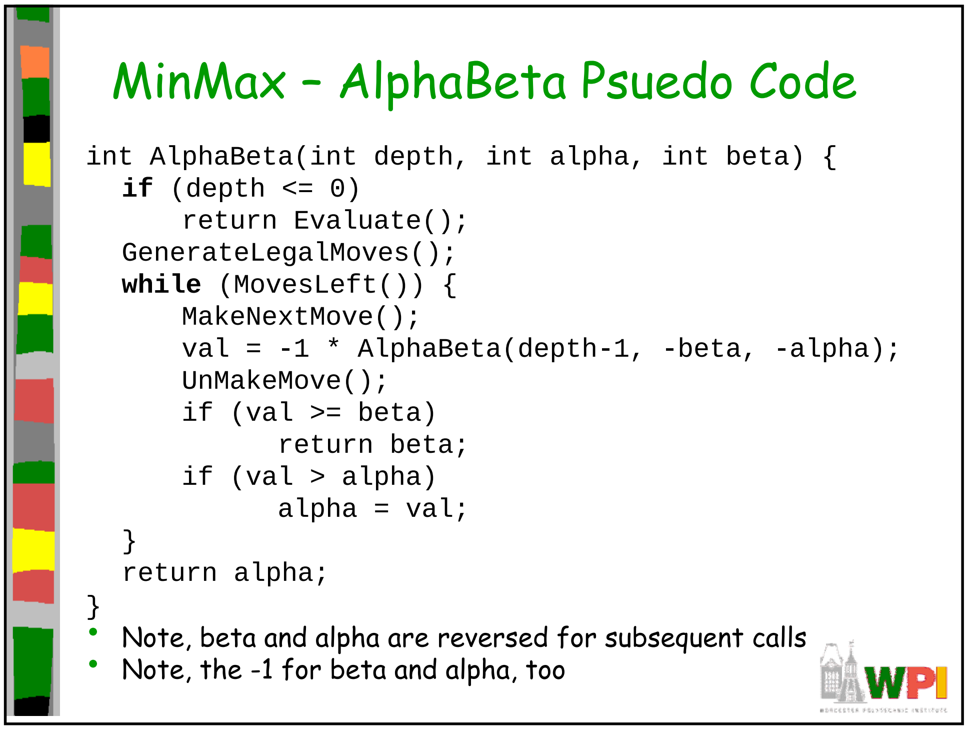


- From Max point of view, 1 is already lower than 4 or 5, so no need to evaluate 2 and 3 (bottom right) → Prune



MinMax – AlphaBeta Pruning Idea

- Two scores passed around in search
  - *Alpha* – best score by some means
    - Anything less than this is no use (can be pruned) since we can already get alpha
    - Minimum score Max will get
    - Initially, negative infinity
  - *Beta* – worst-case scenario for opponent
    - Anything higher than this won't be used by opponent
    - Maximum score Min will get
    - Initially, infinity
- Recursion progresses, the "window" of Alpha-Beta becomes smaller
  - Beta < Alpha → current position not result of best play and can be pruned



MinMax – AlphaBeta Psuedo Code

```
int AlphaBeta(int depth, int alpha, int beta) {
  if (depth <= 0)
      return Evaluate();
  GenerateLegalMoves();
  while (MovesLeft()) {
      MakeNextMove();
      val = -1 * AlphaBeta(depth-1, -beta, -alpha);
      UnMakeMove();
      if (val >= beta)
            return beta;
      if (val > alpha)
            alpha = val;
  }
  return alpha;
}
```

- Note, beta and alpha are reversed for subsequent calls
- Note, the -1 for beta and alpha, too

MinMax – AlphaBeta Notes

- Benefits heavily dependent upon order searched
  - If always start at worst, never prune
    - Ex: consider previous with node 1 first (worst)
  - If always start at best, branch at approximated sqrt(branch)
    - Ex: consider previous with 5 first (best)
- For Chess:
  - If ~35 choices per ply, at best can improve from 35 to 6

    → Allows search twice as deep

MinMax – Notes

- Chess has many forced tactical situations (ie- taken knight, better take other knight)
  - MinMax can leave hanging (at tree depth)
  - So, when done, check for captures only
- Time to search can vary (depending upon `Evaluate()` and branches and pruning)
  - Instead, search 1 ply. Check time. If enough, search 2 plies. Repeat. Called *iterative deepening*

```
depth = 1;
while (1) {
    Val = AlphaBeta(depth, -INF, INF)
    If (timeOut()) break;
}
```

  - For enhancement, can pass in best set of moves (line) seen last iteration (principle variation)

MinMax – `Evaluate()`

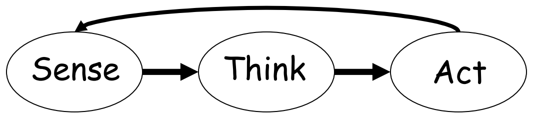
- Checkmate – worth more than rest combined
- Typical, use weighted function:
  - `c1*material + c2*mobility + c3*king safety + c4*center control + ...`
  - Simplest is point value for `material`
    - pawn 1, knight 3, bishop 3, castle 3, queen 9
    - All other stuff worth 1.5 pawns (ie- can ignore most everything else)
- What about a draw?
  - Can be good (ie- if opponent is strong)
  - Can be bad (ie- if opponent is weak)
  - Adjust with *contempt factor*
    - Makes a draw (0) slightly lower (play to win)

Outline

• Introduction	(done)
• MinMax	(done)
• Agents	(next)
• Finite State Machines	
• Common AI Techniques	
• Promising AI Techniques	

Game Agents

- Most AI focuses around game *agent*
  - think of agent as NPC, enemy, ally or neutral
- Loops through: *sense-think-act* cycle
  - Acting is event specific, so talk about *sense* and *think* first, then a bit on *act*



Game Agents – Sensing (1 of 2)

- Gather current world state: barriers, opponents, objects, …
- Needs limitations: avoid "cheating" by looking at game data
  - Typically, same constraints as player (vision, hearing range, etc.)
- **Vision**
  - Can be quite complicated (CPU intensive) to test visibility (ie- if only part of an object visible)
  - Compute vector to each object
    - Check magnitude (ie- is it too far away?)
    - Check angle (dot product) (ie- within 120° viewing angle?)
    - Check if obscured. Most expensive, so do last.

Game Agents – Sensing (2 of 2)

- **Hearing**
  - Ex- tip-toe past, enemy doesn't hear, but if run past, enemy hears (stealth games, like *Thief*)
  - Implement as event-driven
    - When player performs action, notify agents within range
      - Rather than sound reflection (complicated) usually distance within bounded area
    - Can enhance with listen attributes by agent (if agent is "keen eared" or paying attention)
- **Communication**
  - Model sensing data from other agents
  - Can be instant (ie- connected by radio)
  - Or via hearing (ie- shout)
- **Reaction times**
  - Sensing may take some time (ie- don't have agent react to alarm instantly, seems unrealistic)
  - Build in delay. Implement with simple timer.

Game Agents – Thinking (1 of 3)

- Evaluate information and make decision
- As simple or elaborate as required
- Generally, two ways:

1. Pre-coded expert knowledge
   - Typically hand-crafted "if-then" rules + "randomness" to make unpredictable
2. Search algorithm for best (optimal) solution
   - Ex- MinMax

Game Agents – Thinking (2 of 3)

- Expert Knowledge
  - Finite State Machines, decision trees, … (FSM most popular, details next)
  - Appealing since simple, natural, embodies common sense and knowledge of domain
    - Ex: See enemy weaker than you? → Attack. See enemy stronger? → Go get help
  - Trouble is, often does not scale
    - Complex situations have many factors
    - Add more rules, becomes brittle
  - Still, often quite adequate for many AI tasks
    - Many agents have quite narrow domain, so doesn't matter

Game Agents – Thinking (3 of 3)

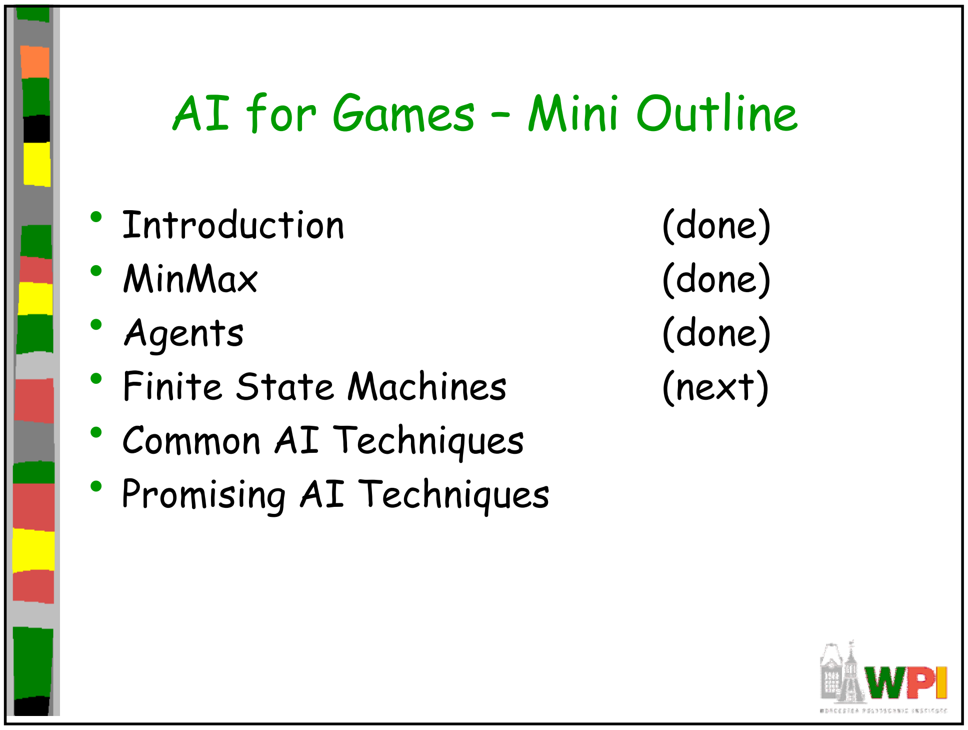
- Search
  - Look ahead and see what move to do next
    - Ex: piece on game board (MinMax), pathfinding (A*)
  - Works well with known information (ie- can see obstacles, pieces on board)
- Machine learning
  - Evaluate past actions, use for future action
  - Techniques show promise, but typically too slow

Game Agents – Acting (1 of 2)

- Learning and Remembering
  - May not be important in many games where agent short-lived (ie- enemy drone)
  - But if alive for 30+ seconds, can be helpful
    - ie- player attacks from right, so shield right
  - Implementation - too avoid too much information, can have fade from memory (by time or by queue that becomes full)

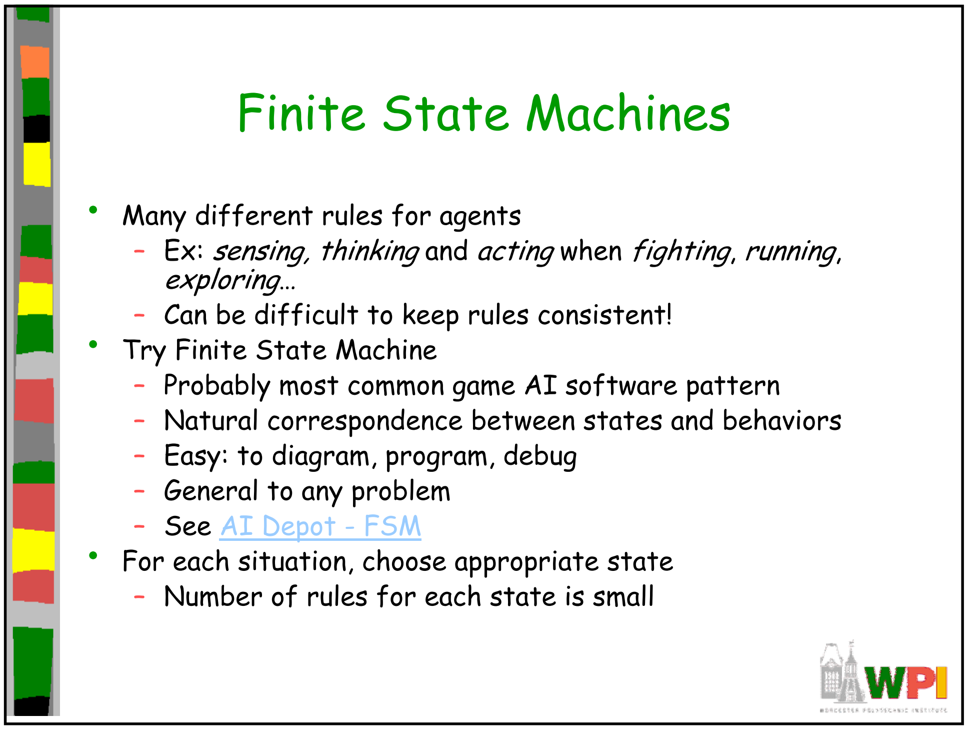
Game Agents – Acting (2 of 2)

- Making agents stupid
  - Many cases, easy to make agents dominate
    - Ex: FPS bot always makes head-shot
  - Dumb down by giving "human" conditions, longer reaction times, make unnecessarily vulnerable, have make mistakes
- Agent cheating
  - Ideally, don't have unfair advantage (such as more attributes or more knowledge)
  - But sometimes might "cheat" to make a challenge
    - Remember, that's the goal, AI lose in challenging way
  - Best to let player know



AI for Games – Mini Outline

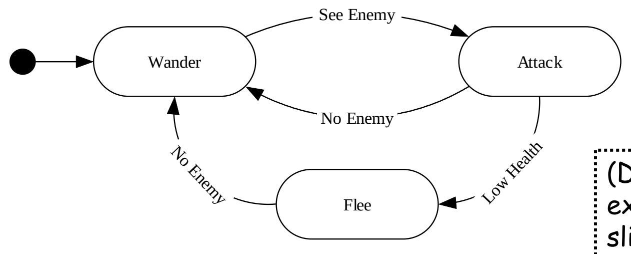
• Introduction	(done)
• MinMax	(done)
• Agents	(done)
• Finite State Machines	(next)
• Common AI Techniques	
• Promising AI Techniques	



Finite State Machines

- Many different rules for agents
  - Ex: *sensing, thinking* and *acting* when *fighting, running, exploring…*
  - Can be difficult to keep rules consistent!
- Try Finite State Machine
  - Probably most common game AI software pattern
  - Natural correspondence between states and behaviors
  - Easy: to diagram, program, debug
  - General to any problem
  - See AI Depot - FSM
- For each situation, choose appropriate state
  - Number of rules for each state is small

Finite State Machines

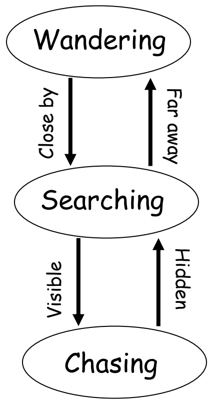


(Do detailed example next slide)

- Abstract model of computation
- Formally:
  - Set of states
  - A starting state
  - An input vocabulary
  - A transition function that maps inputs and the current state to a next state

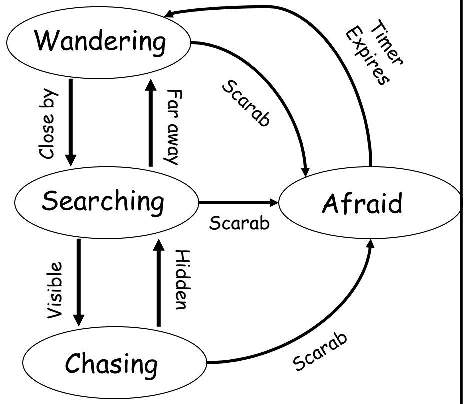
Finite State Machines – Example (1 of 2)

- Game where raid Egyptian Tomb
- Mummies! Behavior
  - Spend all of eternity *wandering* in tomb
  - When player is close, *search*
  - When see player, *chase*
- Make separate states
  - Define behavior in each state
    - Wander – move slowly, randomly
    - Search – move faster, in lines
    - Chasing – direct to player
- Define transitions
  - Close is 100 meters (smell/sense)
  - Visible is line of sight



Finite State Machines – Example (2 of 2)

- Can be extended easily
- Ex: Add magical scarab (amulet)
- When player gets scarab, Mummy is afraid. Runs.
- Behavior
  - Move away from player fast
- Transition
  - When player gets scarab
  - When timer expires
- Can have sub-states
  - Same transitions, but different actions
    - ie- range attack versus melee attack



Finite-State Machine: Approaches

- Three approaches
  - Hardcoded (`switch` statement)
  - Scripted
  - Hybrid Approach

Finite-State Machine: Hardcoded FSM

```
void Step(int *state) { // call by reference since state can change
    switch(state) {

        case 0:  // Wander
            Wander();
            if( SeeEnemy() )    { *state = 1; }
            break;

        case 1:  // Attack
            Attack();
            if( LowOnHealth() ) { *state = 2; }
            if( NoEnemy() )     { *state = 0; }
            break;

        case 2:  // Flee
            Flee();
            if( NoEnemy() )     { *state = 0; }
            break;
    }
}
```

Finite-State Machine: Problems with switch FSM

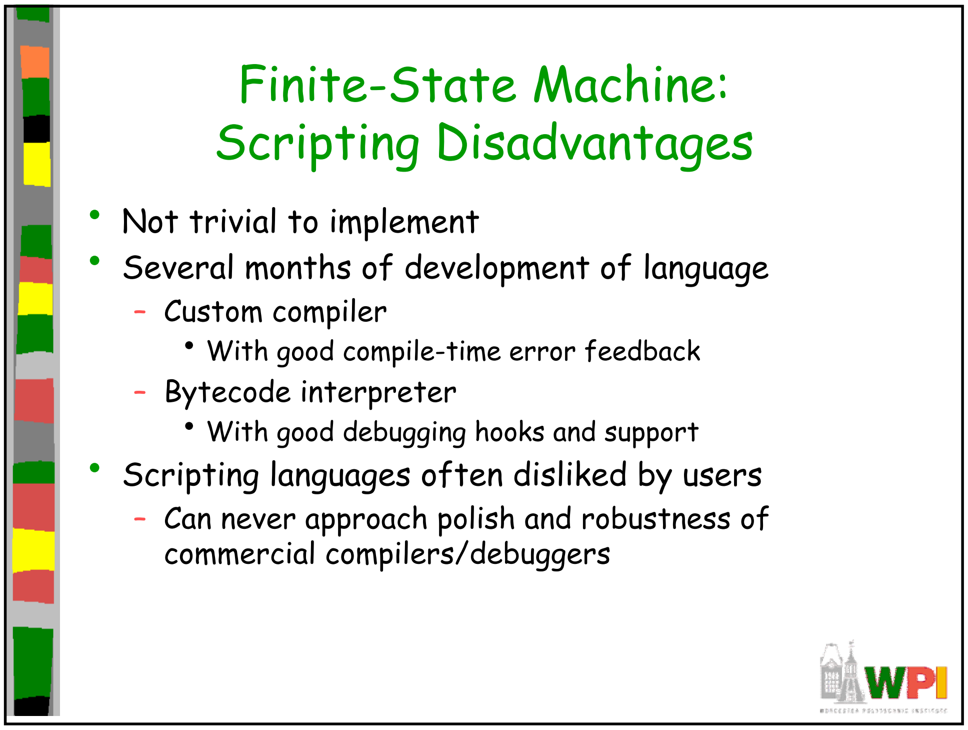
1. Code is ad hoc
   - Language doesn't enforce structure
2. Transitions result from polling (checking each time)
   - Inefficient – event-driven sometimes better
     - ie- when damage, call "pain" event for monster and it may change states
3. Can't determine 1st time state is entered
4. Can't be edited or specified by game designers or players

Finite-State Machine: Scripted with Alternative Language

```
AgentFSM {
    State( STATE_Wander )
        OnUpdate
            Execute( Wander )
            if( SeeEnemy )    SetState( STATE_Attack )
        OnEvent( AttackedByEnemy )
            SetState( Attack )
    State( STATE_Attack )
        OnEnter
            Execute( PrepareWeapon )
        OnUpdate
            Execute( Attack )
            if( LowOnHealth ) SetState( STATE_Flee )
            if( NoEnemy )     SetState( STATE_Wander )
        OnExit
            Execute( StoreWeapon )
    State( STATE_Flee )
        OnUpdate
            Execute( Flee )
            if( NoEnemy )     SetState( STATE_Wander )
}
```

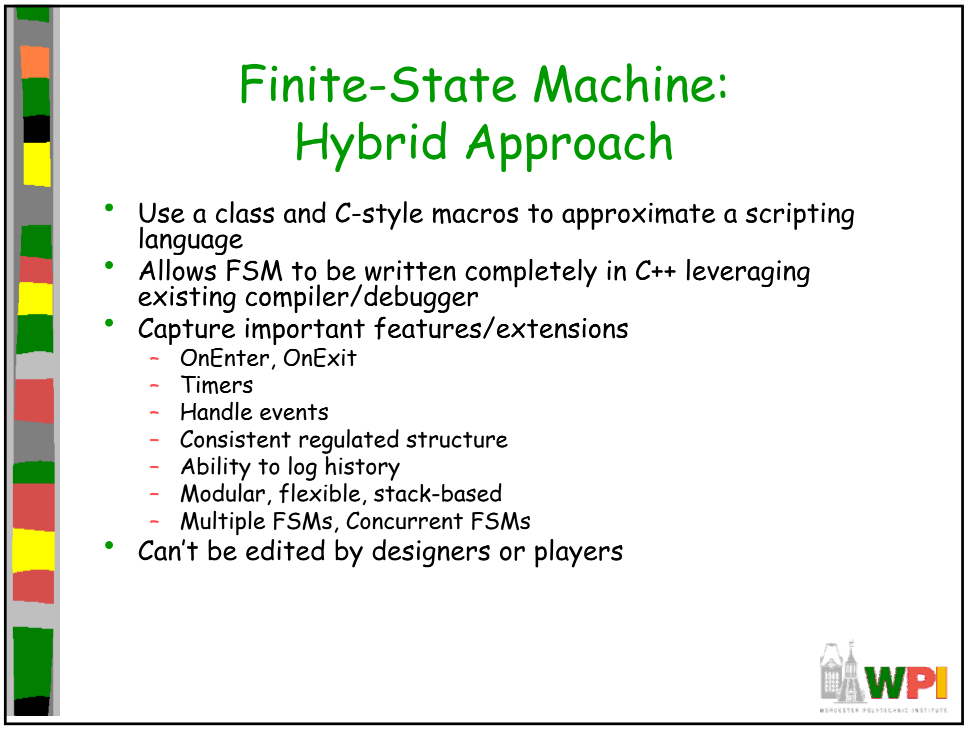
Finite-State Machine: Scripting Advantages

1. Structure enforced
2. Events can be handed as well as polling
3. OnEnter and OnExit concept exists
   (If objects, when created or destroyed)
4. Can be authored by game designers
   - Easier learning curve than straight C/C++



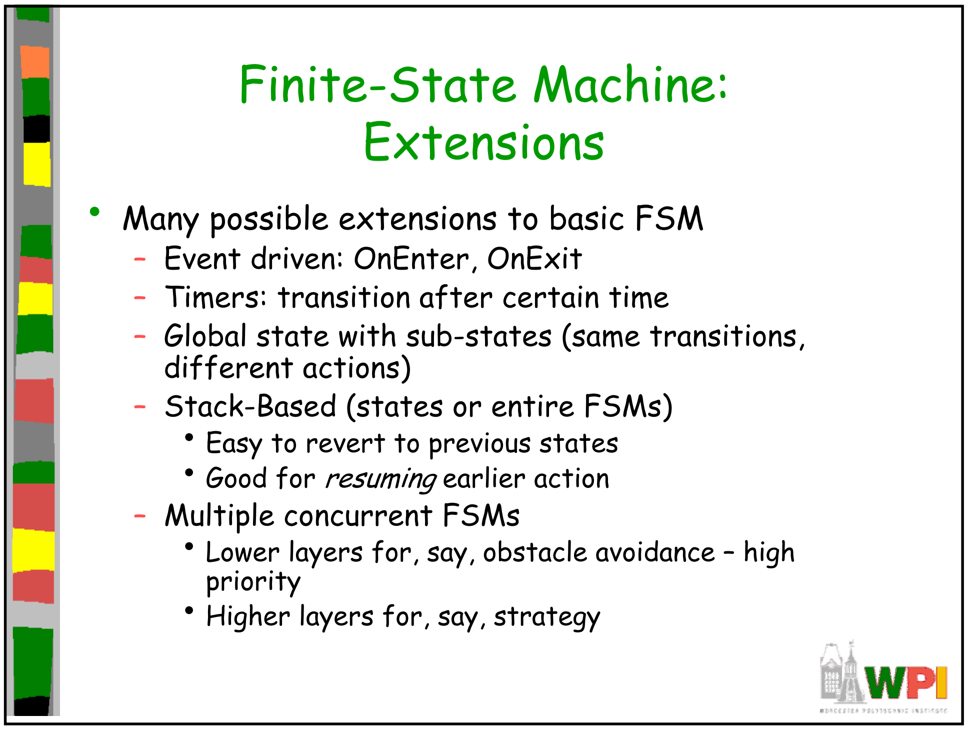
Finite-State Machine: Scripting Disadvantages

- Not trivial to implement
- Several months of development of language
  - Custom compiler
    - With good compile-time error feedback
  - Bytecode interpreter
    - With good debugging hooks and support
- Scripting languages often disliked by users
  - Can never approach polish and robustness of commercial compilers/debuggers



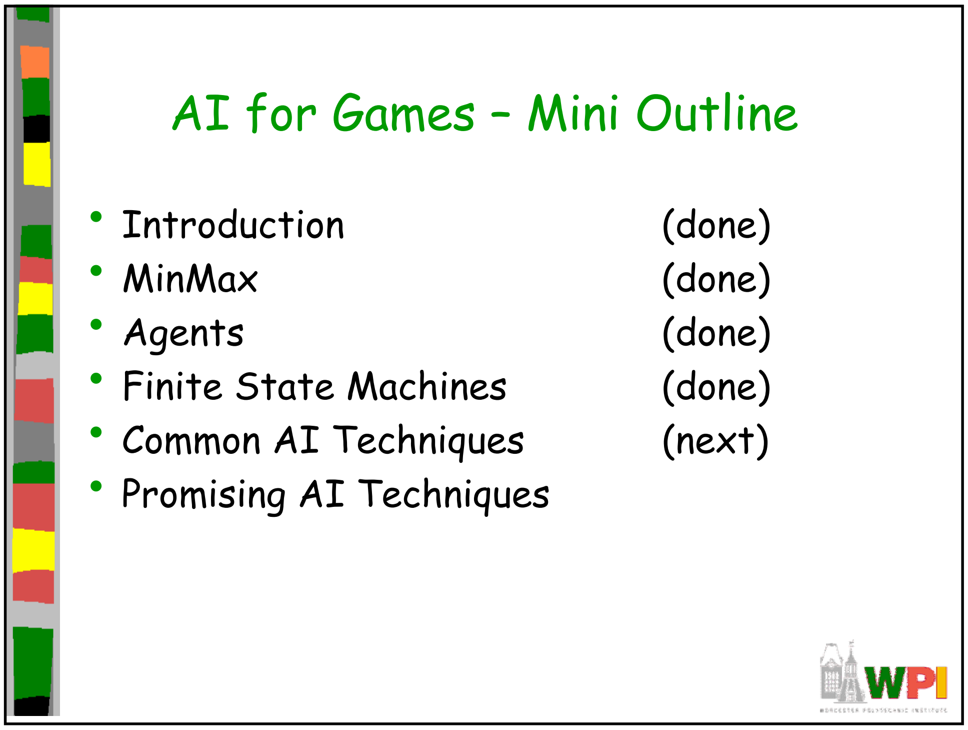
Finite-State Machine: Hybrid Approach

- Use a class and C-style macros to approximate a scripting language
- Allows FSM to be written completely in C++ leveraging existing compiler/debugger
- Capture important features/extensions
  - OnEnter, OnExit
  - Timers
  - Handle events
  - Consistent regulated structure
  - Ability to log history
  - Modular, flexible, stack-based
  - Multiple FSMs, Concurrent FSMs
- Can't be edited by designers or players



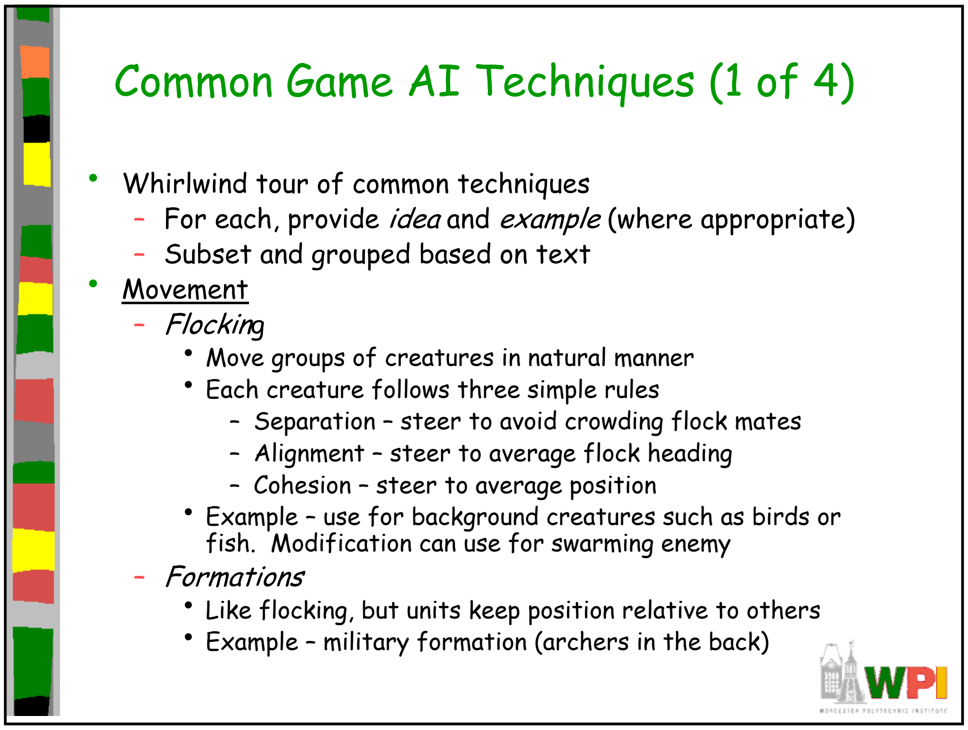
Finite-State Machine: Extensions

- Many possible extensions to basic FSM
  - Event driven: OnEnter, OnExit
  - Timers: transition after certain time
  - Global state with sub-states (same transitions, different actions)
  - Stack-Based (states or entire FSMs)
    - Easy to revert to previous states
    - Good for *resuming* earlier action
  - Multiple concurrent FSMs
    - Lower layers for, say, obstacle avoidance – high priority
    - Higher layers for, say, strategy



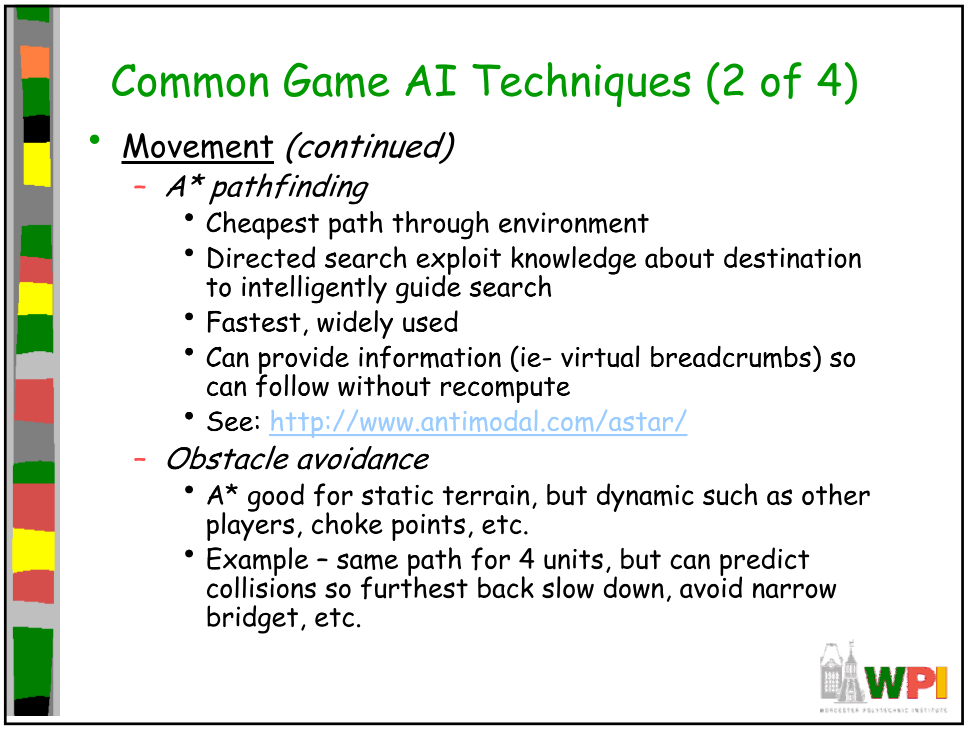
AI for Games – Mini Outline

• Introduction	(done)
• MinMax	(done)
• Agents	(done)
• Finite State Machines	(done)
• Common AI Techniques	(next)
• Promising AI Techniques	



Common Game AI Techniques (1 of 4)

- Whirlwind tour of common techniques
  - For each, provide *idea* and *example* (where appropriate)
  - Subset and grouped based on text
- Movement
  - *Flocking*
    - Move groups of creatures in natural manner
    - Each creature follows three simple rules
      - Separation – steer to avoid crowding flock mates
      - Alignment – steer to average flock heading
      - Cohesion – steer to average position
    - Example – use for background creatures such as birds or fish. Modification can use for swarming enemy
  - *Formations*
    - Like flocking, but units keep position relative to others
    - Example – military formation (archers in the back)



Common Game AI Techniques (2 of 4)

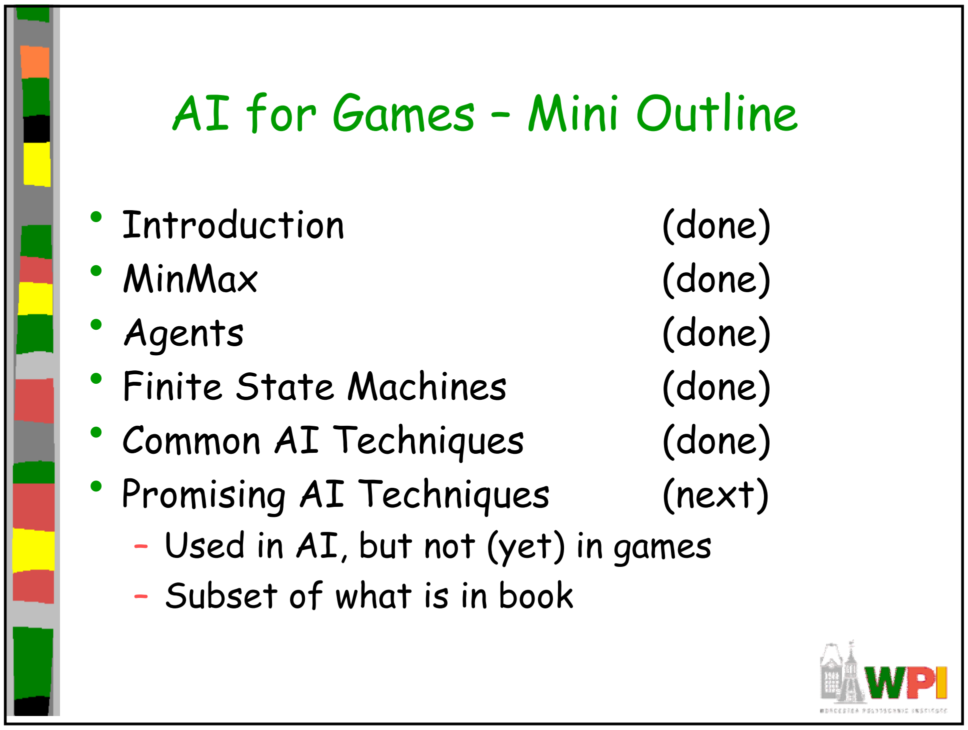
- Movement *(continued)*
  - *A* pathfinding*
    - Cheapest path through environment
    - Directed search exploit knowledge about destination to intelligently guide search
    - Fastest, widely used
    - Can provide information (ie- virtual breadcrumbs) so can follow without recompute
    - See: http://www.antimodal.com/astar/
  - *Obstacle avoidance*
    - A* good for static terrain, but dynamic such as other players, choke points, etc.
    - Example – same path for 4 units, but can predict collisions so furthest back slow down, avoid narrow bridget, etc.

Common Game AI Techniques (3 of 4)

- Behavior organization
  - *Emergent behavior*
    - Create simple rules result in complex interactions
    - Example: game of life, flocking
  - *Command hierarchy*
    - Deal with AI decisions at different levels
    - Modeled after military hierarchy (ie- General does strategy to Foot Soldier does fighting)
    - Example: Real-time or turn based strategy games -- overall strategy, squad tactics, individual fighters
  - *Manager task assignment*
    - When individual units act individually, can perform poorly
    - Instead, have manager make tasks, prioritize, assign to units
    - Example: baseball – 1st priority to field ball, 2nd cover first base, 3rd to backup fielder, 4th cover second base. All players try, then disaster. Manager determines best person for each. If hit towards 1st and 2nd, first baseman field ball, pitcher cover first base, second basemen cover first

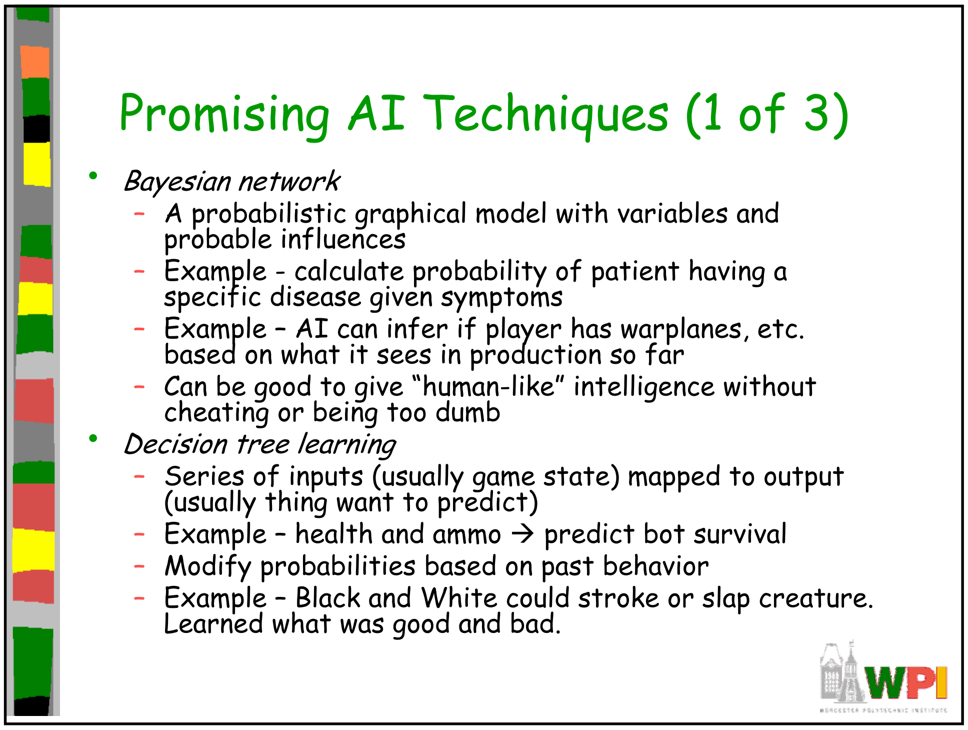
Common Game AI Techniques (4 of 4)

- *Influence map*
  - 2d representation of power in game
  - Break into cells, where units in each cell are summed up
  - Units have influence on neighbor cells (typically, decrease with range)
  - Insight into location and influence of forces
  - Example – can be used to plan attacks to see where enemy is weak or to fortify defenses. SimCity used to show fire coverage, etc.
- *Level of Detail AI*
  - In graphics, polygonal detail less if object far away
  - Same idea in AI – computation less if won't be seen
  - Example – vary update frequency of NPC based on position from player



AI for Games – Mini Outline

- Introduction (done)
- MinMax (done)
- *Agents* (done)
- Finite State Machines (done)
- Common AI Techniques (done)
- Promising AI Techniques (next)
  - Used in AI, but not (yet) in games
  - Subset of what is in book



Promising AI Techniques (1 of 3)

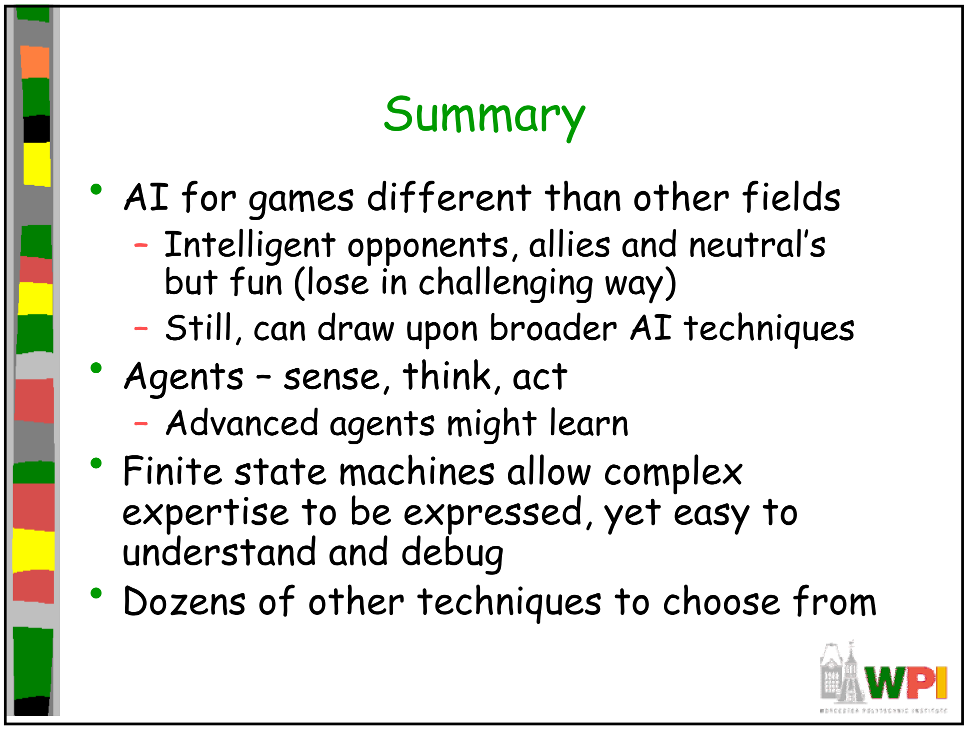
- *Bayesian network*
  - A probabilistic graphical model with variables and probable influences
  - Example - calculate probability of patient having a specific disease given symptoms
  - Example – AI can infer if player has warplanes, etc. based on what it sees in production so far
  - Can be good to give "human-like" intelligence without cheating or being too dumb
- *Decision tree learning*
  - Series of inputs (usually game state) mapped to output (usually thing want to predict)
  - Example – health and ammo → predict bot survival
  - Modify probabilities based on past behavior
  - Example – Black and White could stroke or slap creature. Learned what was good and bad.

Promising AI Techniques (2 of 3)

- *Filtered randomness*
  - Want randomness to provide unpredictability to AI
  - But even random can look odd (ie- if 4 heads in a row, player think something wrong. And, if flip coin 100 times, will be streak of 8)
    - Example – spawn at same point 5 times in a row, then bad
  - Compare random result to past history and avoid
- *Fuzzy logic*
  - Traditional set, object belongs or not.
  - In fuzzy, can have relative membership (ie- hungry, not hungry. Or "in-kitchen" or "in-hall" but what if on edge?)
  - Cannot be resolved by coin-flip
  - Can be used in games – ie- assess relative threat

Promising AI Techniques (3 of 3)

- *Genetic algorithms*
  - Search and optimize based on evolutionary principles
  - Good when "right" answer not well-understood
  - Example – may not know best combination of AI settings. Use GA to try out
  - Often expensive, so do offline
- *N-Gram statistical prediction*
  - Predict next value in sequence (ie- 1818180181 … next will probably be 8)
  - Search backward n values (usually 2 or 3)
  - Example
    - Street fighting (punch, kick, low punch…)
    - Player does low kick and then low punch. What is next?
    - Uppercut 10 times (50%), low punch (7 times, 35%), sideswipe (3 times, 15%)
    - Can predict uppercut or, proportionally pick next (ie- roll dice)



Summary

- AI for games different than other fields
  - Intelligent opponents, allies and neutral's but fun (lose in challenging way)
  - Still, can draw upon broader AI techniques
- Agents – sense, think, act
  - Advanced agents might learn
- Finite state machines allow complex expertise to be expressed, yet easy to understand and debug
- Dozens of other techniques to choose from