Basic Game AI

Technical Game Development II

Professor Charles Rich
Computer Science Department
rich@wpi.edu

Definitions?

- What is artificial intelligence (AI)?
  - subfield of computer science?
  - subfield of cognitive science?

- What is “AI for Games”?
  - versus “academic AI”?
  - arguments about “cheating”

In games, everything (including the AI) is in service of the player’s experience (“fun”)

What’s the AI part of a game?

- Everything that isn’t graphics (sound) or networking... 😊
  - or physics (though sometimes lumped in)
  - usually via the non-player characters
  - but sometimes operates more broadly, e.g.,
    - Civilization games
    - interactive storytelling

“Levels” of Game AI

- **Basic**
  - decision-making techniques commonly used in almost all games

- **Advanced**
  - used in practice, but in more sophisticated games

- **Future**
  - not yet used, but explored in research
This course

- **Basic game AI** (this week)
  - decision-making techniques commonly used in almost all games
    - decision trees
    - (hierarchical) state machines
    - scripting
    - minimax search
    - pathfinding (beyond A*)

- **Advanced game AI** (weeks 5-6)
  - used in practice, but in more sophisticated games
    - autonomous movement, steering (3 lectures)
    - goal-based AI in Halo 3 (2 lectures from GDC)

Future Game AI?

- Take IMGD 400X next year (B)
  - “AI for Interactive Media and Games”
    - fuzzy logic
    - more goal-driven agent behavior

- Take CS 4341 “Artificial Intelligence”
  - machine learning
  - planning
Two Fundamental Types of AI Algorithms

- **Search vs. Non-Search**
  - *non-search*: amount of computation is predictable (decision trees, state machines)
  - *search*: upper bound depends on size of search space (often large)
    - scary for real-time games
    - need to otherwise limit computation (e.g., threshold)

- **Where’s the “knowledge”?**
  - *non-search*: in the code logic (or external tables)
  - *search*: in state evaluation and search order functions

First Basic AI Technique:

**Decision Trees**

*Reference: Millington, Section 5.2*
Decision Trees

- The most basic of the basic AI techniques
- Easy to implement
- Fast execution
- Simple to understand

Deciding how to respond to an enemy

```java
if (visible) {
    if (close) {
        attack;
    } else {
        if (flank) {
            move;
        } else {
            attack;
        }
    }
} else {
    if (audible) {
        creep;
    }
}
```
Which would you rather modify?

```java
if (visible) {
    if (close) {
        attack;
    } else if (flank) {
        move;
    } else {
        attack;
    }
} else if (audible) {
    creep;
}
```

Designing OO Decision Trees

(see Millington, Section 5.2.3)

```python
class Node:
    def decide()

class Action(Node):
    def decide()
        return this

class Decision(Node):
    yesNode
    noNode
    testValue
    getBranch()
    def decide()
        return getBranch().decide()

    class MinMax(Decision):
        minValue
        maxValue
        def getBranch()
            if maxValue >= testValue >= minValue
                return yesNode
            else
                return noNode
```
Building and Maintaining a Decision Tree

visible = decision[0] = new Boolean...
audible = decision[1] = new Boolean...
close = decision[2] = new MinMax...
flank = decision[3] = new Boolean...

attack = action[0] = new Move...
move = action[1] = new Move...
creep = action[2] = new Creep...

visible.yesNode = close
visible.noNode = audible
audible.yesNode = creep
close.yesNode = attack
close.noNode = flank
flank.yesNode = move
flank.noNode = attack

...or a graphical editor

Performance Issues

- individual node tests (getBranch) typically constant time (and fast)
- worst case behavior depends on depth of tree
  - longest path from root to action
- roughly “balance” tree (when possible)
  - not too deep, not too wide
  - make commonly used paths shorter
  - put most expensive decisions late
Next Basic AI Technique:

(Hierarchical) State Machines

References: Buckland, Chapter 2  
Millington, Section 5.3

State Machines
Hard-Coded Implementation

class Soldier
enum State
    GUARD
    FIGHT
    RUN_AWAY
currentState

def update()
    if currentState = GUARD {
        if (small enemy)
            currentState = FIGHT
            startFighting
        if (big enemy)
            currentState = RUN_AWAY
            startRunningAway
    } else if currentState = FIGHT {
        if (losing fight) c
            currentState = RUN_AWAY
            startRunningAway
    } else if currentState = RUN_AWAY {
        if (escaped)
            currentState = GUARD
            startGuarding
    }

Hard-Coded State Machines

- Easy to write (at the start)
- Very efficient
- Notoriously hard to maintain (e.g., debug)
Cleaner & More Flexible Implementation

```python
class State:
    def __init__(self):
        pass

    def getAction(self):
        pass

    def getEntryAction(self):
        pass

    def getExitAction(self):
        pass

    def getTransitions(self):
        pass

class StateMachine:
    states = [initialState for initialState in States]
    currentState = initialState

    def update(self):
        triggeredTransition = None
        for transition in self.currentState.getTransitions():
            if transition.isTriggered():
                triggeredTransition = transition
                break
        if triggeredTransition:
            self.currentState = triggeredTransition.getTargetState()
            actions = self.currentState.getExitAction()
            actions += triggeredTransition.getAction()
            actions += self.currentState.getEntryAction()
        else:
            return self.currentState.getAction()

...add tracing
```

(see Millington, Section 5.3.3)

Combining Decision Trees & State Machines

- Why?
  - to avoid duplicating expensive tests

![Combining Decision Trees & State Machines Diagram](image)
Combining Decision Trees & State Machines

Hierarchical State Machines

- Why?

Diagram:

- Alert
  - player in sight?
    - yes → alarm
    - no → far?
      - yes → defend
      - no → yes
-
- search
  - see trash
    - trash disposed
      - trash disposed
    - have trash
      - goto disposal
Interruptions (Alarms)

6 - doubled the number of states!

Add Another Interruption Type

12 - doubled the number of states again!
Hierarchical State Machine

- leave any state in (composite) ‘clean’ state when ‘low power’
- ‘clean’ remembers internal state and continues when returned to via ‘recharged’

Add Another Interruption Type

7 states (including composite) vs. 12
Cross-Hierarchy Transitions

Why?
- suppose we want robot to top-off battery if it doesn't see any trash

Cross-Hierarchy Transitions

Cross-Hierarchy Transitions

less than 75% power

Cross-Hierarchy Transitions

Cross-Hierarchy Transitions
Implementation Sketch

class State
    # stack of return states
    def getStates() return [this]
    # recursive update
    def update()
    # rest same as flat machine

class Transition
    # how deep this transition is
    def getLevel()
    # rest same as flat machine

struct UpdateResult # returned from update
    transition
    level
    actions # same as flat machine

class HierarchicalStateMachine
    # same state variables as flat machine
    # complicated recursive algorithm
    def update()

class SubMachine : State,
    HierarchicalStateMachine
    def getStates()
        push [this] onto currentState.getStates()

(see Millington, Section 5.3.9)

Next Basic AI Technique:

Scripting

References:  Buckland, Chapter 6
Millington, Section 5.9
AI Scripting

Has something to do with:

- scripting languages
- role of scripting in the game development process

Scripting Languages

You can probably name a bunch of them:

- general purpose languages
  - Tcl, Python, Perl, Javascript, Ruby, Lua, ...
- tied to specific games/engines
  - UnrealScript, QuakeC, HaloScript, LSL, ...
What makes a general purpose scripting language different from any other programming language?

- interpreted (byte code, virtual machine)
  - faster development cycle
  - safely executable in “sandbox”
- simpler syntax/semantics:
  - untyped
  - garbage-collected
  - builtin associative data structures
- plays well with other languages
  - e.g., LiveConnect, .NET

But when all is said and done, it looks pretty much like “code” to me....

e.g. Lua

    function factorial(n)
        if n == 0 then
            return 1
        end
        return n * factorial(n - 1)
    end
General Purpose Scripting Languages

So it must be about something else...

Namely, the game development process:

- For the technical staff
  - data-driven design (scripts viewed as data, not part of codebase)
  - script changes do not require game recompilation
- For the non-technical staff
  - allows parallel development by designers
  - allows end-user extension

But to make this work, you need to successfully address a number of issues:

- Where to put boundaries (APIs) between scripted and “hard-coded” parts of game
- Performance
- Flexible and powerful debugging tools
  - even more necessary than with some conventional (e.g., typed) languages
- Is it really easy enough to use for designers!?
Lua in Games

* Aleph One (an open-source enhancement of Marathon 2: Durandal) supports Lua, and it's been used in a number of scenarios (including Excalibur and Eternal).
* Blobby Volley, in which bots are written in Lua.
* Company of Heroes, a WW2 RTS. Lua is used for the console, AI, single player scripting, win condition scripting and for storing unit attributes and configuration information.
* Crysis, a first-person shooter & spiritual successor to Far Cry.
* Dawn of War, uses Lua throughout the game.
* Destroy All Humans! and Destroy All Humans! 2 both use Lua.
* Escape from Monkey Island is coded in Lua instead of the SCUMM engine of the older titles. The historic "SCUMM Bar" is renovated and renamed to the "Lua Bar" as a reference.
* Far Cry, a first-person shooter. Lua is used to script a substantial chunk of the game logic, manage game objects (Entity system), configure the HUD and store other configuration information.
* Garry's Mod and Fortress Forever, mods for Half-Life 2, use Lua scripting for tools and other sorts of things for full customization.
* Grim Fandango and Escape from Monkey Island, both based on the GrimE engine, are two of the first games which used Lua for significant purposes.

Lua in Games (cont’d)

* Gusanos (Version 0.9) supports Lua Scripting for making the whole game modable.
* Homeworld 2 uses Lua scripting for in-game levels, AI, and as a Rules Engine for game logic.
* Incredible Hulk: Ultimate Destruction uses Lua for all mission scripting
* JKALua, A game modification for the game JK3: Jedi Academy.
* Multi Theft Auto, a multi-player modification for the Grand Theft Auto video game series. The recent adaptation for the game Grand Theft Auto San Andreas uses Lua.
* Painkiller
* Ragnarok Online recently had a Lua implementation, allowing players to fully customize the artificial intelligence of their homunculus to their liking, provided that they have an Alchemist to summon one.
* ROBLOX is an online Lego-like building game that uses Lua for all in-game scripting.
* SimCity 4 uses Lua for some in-game scripts.
* Singles: Flirt Up Your Life uses Lua for in-game scripts and object/character behavior.
* Spring (computer game) is an advanced open-source RTS engine, which is able to use Lua for many things, including unit/mission scripting, AI writing as well as interface changes.
* S.T.A.L.K.E.R.: Shadow of Chernobyl
* Star Wars: Battlefront and Star Wars: Battlefront 2 both use Lua.
Lua in Games (cont’d)

- Star Wars: Empire at War uses Lua.
- Supreme Commander allows you to edit almost all its aspects with Lua.
- Toribash, a turn-based fighting game, supports Lua scripting.
- Vendetta Online, a science fiction MMORPG, lets users use Lua to customize the user interface, as well as create new commands and react to events triggered by the game.
- Warhammer Online uses Lua.
- The Witcher.
- World of Warcraft, a fantasy MMORPG. Lua is used to allow users to customize its user interface.
- Xmoto, a free and open source 2D motocross platform game, supports Lua scripting in levels.

The Other Path...

- A custom scripting language tied to a specific game, which is just idiosyncratically “different” (e.g., QuakeC) doesn’t have much to recommend it.
- However, a game-specific scripting language that is truly natural for non-programmers can be very effective:

```plaintext
if enemy health < 500 && enemy distance < our bigrange
  move ...
  fire ...
else
  ...
return

(GalaxyHack)
```
Custom Tools with Integrated Scripting

“Designer UI” from Halo 3

Next Basic AI Technique:
Minimax Search

Reference: Millington, Section 8.2
**Minimax Search**

- Minimax is at 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 with chance
  - Poker, Monopoly, Dice
- Can work in real-time (i.e., not turn based) with timer (*iterative deepening*, later)

**The Game Tree**

E.g., Tic-Tac-Toe

*Note:* Just showing top part of tree
- Symmetrical positions removed (optimization example)
### The Game Tree

- Nodes in tree represent *states*
  - e.g., board configurations, “positions”
- Arcs are *decisions* that take you to a next state
  - e.g., “moves”
- Technically a *directed acyclic graph*
  - may have joins but no cycles
- Levels called *plies* (plural of *ply*)
  - players alternate levels (or rotate among >2 players)

---

### Naive Approach

1. Exhaustively expand tree
   - naive because tree may be too big
   - e.g., chess
     - typical board position has ~35 legal moves
     - for 40 move game, $35^{40}$ > number atoms in universe
2. Choose next move on a path that leads to your winning
   - assumes your opponent is going to cooperate and “let” you win
   - on his turn, he most likely will choose the *worst* case for you!
Minimax Approach

- assume both/all players play to the best of their ability
- define a scoring method (see next)
- from the standpoint of a given player (let’s call him “Max” 😊):
  - choose move which takes you to the next state with highest expected score (from your point of view)
  - assuming the other player (let’s call her “Minnie” 😜) will on her move choose the next state with the lowest score (from your point of view)

(Static) Evaluation Function

- assigns score to given state from point of view of given player
  - scores typically integers in centered range
    - e.g., [-100,+100] for TTT
    - e.g., [-1000,+1000] for chess
  - extreme values reserved for win/lose
    - this is typically the easy case to evaluate
    - e.g., for first player in TTT, return +100 if board has three X’s in a row or -100 if three O’s in a row
    - e.g., checkmate for chess
  - what about non-terminal states?
(Static) Evaluation Function

- much harder to score in middle of the game
- score should reflect “likelihood” a player will win from given state (board position)
- but balance of winning/losing isn’t always clear (e.g., number/value of pieces, etc.)
  - e.g., in Reversi, best strategy is to have fewest counters in middle of game (better board control)
  - generic “local maxima” problem with all “hill climbing” search methods
- static evaluation function is where (most) game-specific knowledge resides

Naive Approach

1. Apply static evaluation to each next state
2. Choose move to highest scoring state

*If static evaluation function were perfect, then this is all you need to do*
  - perfect static evaluator almost never exists
  - using this approach with imperfect evaluator performs very badly

*The solution?* Look ahead!
Minimax Looking Ahead

- It’s Max’s turn at the start of the game (root of the tree)
- There is only time to expand tree to 2nd ply
- Max’s static evaluation function has been applied to all leaf states
- Max would “like” to get to the 9 point state
- But if chooses leftmost branch, Min will choose her move to get to 3
  → left branch has a value of 3
- If Max chooses rightmost branch, 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

Minimax “Bubbling Up Values”

- Max’s turn (root of tree)
- Circles represent Max’s turn, Squares represent Min’s turn
- Values in leaves are result of applying static evaluation function
- Red arrows represent best (local) move for each player
- Blue arrow is Max’s chosen move on this turn
Minimax Algorithm

```python
def MinMax(board, player, depth, maxDepth):
    if (board.isGameOver() or depth == maxDepth):
        return board.evaluate(player), null

    bestMove = null
    if (board.currentPlayer() == player):
        bestScore = -INFINITY
    else:
        bestScore = +INFINITY

    for move in board.getMoves():
        newBoard = board.makeMove(move)
        score, move = MinMax(newBoard, player, depth+1, maxDepth)
        if (board.currentPlayer() == player):
            if (score > bestScore): # max
                bestScore = score
                bestMove = move
        else:
            if (score < bestScore): # min
                bestScore = score
                bestMove = move

    return bestScore, bestMove

MinMax(board, player, 0, maxDepth)
```

Note: makeMove returns copy of board (can also move/unmove--but don't execute graphics!)

Negamax Version

- for common case of
  - two player
  - zero sum
- single static evaluation function
  - returns + or - same value for given board position, depending on player
### Negamax Algorithm

```python
def Negamax(board, depth, maxDepth):
    if (board.isGameOver() or depth == maxDepth):
        return board.evaluate(), null

    bestMove = null
    bestScore = -INFINITY

    for move in board.getMoves():
        newBoard = board.makeMove(move)
        score, move = Negamax(newBoard, depth+1, maxDepth)
        score = -score
        if (score > bestScore):
            bestScore = score
            bestMove = move

    return bestScore, bestMove

Negamax(board, 0, maxDepth)
```

### Pruning Approach

- Minimax searches entire tree, even if in some cases it is clear that parts of the tree can be ignored (pruned)
- **Example**:
  - You won a bet with your *enemy*.
  - He owes you one thing from a collection of bags.
  - You get to choose the bag, but your *enemy* chooses the thing.
  - Go through the 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 what the rest of the items in this bag are ($500, Yankee’s tickets …)
    - No point in looking further in *this bag*, since enemy’s dead fish is already worse than sandwich
### Pruning Approach

- In general,
- Stop processing branches at a node when you find a branch worse than result you already know you can achieve
- This type of pruning saves processing time *without* affecting final result
  - i.e., *not* a “heuristic” like the evaluation function in A*

### Pruning Example

- From Max’s point of view, 1 is already lower than 4, which he knows he can achieve, so there is no need to look farther at sibling branches
- Note that there might be *large* subtrees below nodes labeled 2 and 3 (only showing the top part of tree)
Alpha-Beta Pruning

- Keep track of two scores:
  - **Alpha** – best score by any 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
- As recursion progresses, the "window" of Alpha-Beta becomes smaller
  - (Beta < Alpha) → current position not result of best play and can be pruned

---

Alpha-Beta NegaMax Algorithm

```python
def ABNegaMax(board, depth, maxDepth, alpha, beta):
    if (board.isGameOver() or depth == maxDepth):
        return board.evaluate(player), null

    bestMove = null
    bestScore = -INFINITY

    for move in board.getMoves():
        newBoard = board.makeMove(move)
        score, move = ABNegaMax(newBoard, maxDepth, depth+1,
                                -beta, -max(alpha, bestScore))

        score = -score
        if (score > bestScore):
            bestScore = score
            bestMove = move

        # early loop exit (pruning)
        if (bestScore >= beta):
            return bestScore, bestMove

    return bestScore, bestMove

ABNegaMax(board, player, maxDepth, 0, -INFINITY, INFINITY)
```
Benefits of pruning depend heavily on order in which branches (moves) are visited
- for example, if branches visited right to left above no pruning happens!
- for chess, on average reduce 35 branches -> 6
  - allows search twice as deep!

Can we improve branch (move) order?
- apply static evaluation function at intermediate nodes and check best first
  - logical idea
  - can improve pruning
  - but may effectively give up depth of search advantage (in fixed time interval) due to high cost of function evaluation
- better idea: use results of previous minimax searches
  - “negascout” algorithm (extra credit, see Millington 8.2.7)
Chess Notes

- Chess has many forced tactical situations
  - e.g., “exchanges” of pieces
  - minimax may not find these
  - add cheap check at end of turn to check for immediate captures
- Library of openings and/or closings
- Use iterative deepening
  - search 1-ply deep, check time, search 2nd ply, ..

Chess Notes

- Static evaluation function
  - typically use weighted function
    - $c_1 \times \text{material} + c_2 \times \text{mobility} + c_3 \times \text{kingSafety} + \ldots$
  - simplest is point value for material
    - pawn 1, knight 3, bishop 3, castle 3, queen 9
  - see references in homework instructions
  - checkmate is worth more than rest combined
  - what about a draw?
    - can be good (e.g., if opponent strong)
    - can be bad (e.g., if opponent weak)
    - adjust with “contempt factor” (above or below zero)
Next Basic AI Technique:

Pathfinding

References: Buckland, Chapter 5, 8
Millington, Chapter 4

---

A* Pathfinding Search

- Covered in IMGD 3000
- Review below if needed

References: Buckland, Chapter 5 (pp. 241-247)
Millington, Section 4.3
Practical Path Planning

- Just raw A* not enough
- Also need:
  - navigation graphs
    - points of visibility (POV)
    - navmesh
  - path smoothing
  - compute-time optimizations
  - hierarchical pathfinding
  - special case methods

Navigation Graph Construction

- Tile (cell) based
  - very common, esp. if env’t already designed in squares or hexagons
  - each cell already labelled with material (mud, etc.)
  - downside:
    - modest 100x100 cell map
    - 10,000 nodes and 78,000 edges
    - can burden CPU and memory, especially if multiple AI’s calling in

Rest of presentation is a survey about how to do better...
Point of Visibility (POV) Navigation Graph

- Place graph nodes (usually by hand) at important points in env’t
- Such that each node has line of sight to at least one other node

NavMesh

- network of convex polygons
- very efficient to search
- can be automatically generated from polygons
- becoming very popular
POV Navigation

- find closest visible node (a) to current location
- find closest visible node (b) to target location
- search for least cost path from (a) to (b)
- move to (a)
- follow path to (b)
- move to target location

POV Navigation

- Obvious how to build and expand
- *Downsides*
  - can take a lot of developer time, especially if design is rapidly evolving
  - problematic if random or user generated maps
  - can have “blind spots”
  - can have “jerky” paths
- *Solutions*
  - automatically generate POV graph from polygons
  - make finer grained graphs
  - smooth paths
Automatic POV by Expanded Geometry

1. expand geometry by amount proportional to bounding radius of agents
2. add vertices to graph
3. prune non line of sight points

Blind Spots in POV

- No POV point is visible from red spots!
- Easy to fix manually in small graphs
- A problem in larger graphs

Solution: finely grained graphs
**Finely Grained Graphs**

- Improves blind spots and path smoothness
- Typically generate automatically using “flood fill”

**Flood Fill**

- Same algorithm as in “paint” programs
Path Finding in Finely Grained Graph

- use A* or Dijkstra depending on whether looking for one or multiple targets

Kinky Paths

*The solution:* Path smoothing
Simple Smoothing Algorithm

- Check for “passability” between adjacent edges

Smoothing Example
Methods to Reduce CPU Overhead

Hierarchical Path Planning

- reduces CPU overhead
- typically two levels, but can be more
- first plan in high-level, then refine in low-level
Getting Out of Sticky Situations

- bot gets “wedged” against wall
- looks really bad!

- Heuristic:
  - calculate the distance to bot’s current waypoint each update step
  - if this value remains about the same or consistently increases
  - then it’s probably wedged
  - backup and replan
Pathfinding Summary

- You would not necessarily use all of these techniques in one game
- Only use whatever your game demands and no more

Basic Game AI Summary

- Decision-making techniques commonly used in almost all games
  - decision trees
  - (hierarchical) state machines
  - scripting
  - minimax search
  - pathfinding (beyond A*)
    - References: Buckland 2, 5, 8; Millington 4, 5, 8
- Advanced game AI (weeks 5-6)
  - used in practice, but in more sophisticated games
    - autonomous movement, steering (3 lectures)
    - goal-based AI in Halo 3 (2 lectures from GDC)