AI in Games

Overview

- A smattering of game AI techniques
  - Goal-based planning
  - Influence maps
  - Neural networks
    - Evolved virtual creatures
    - NERO
    - Learning bots in Quake
  - Modeling randomness

Classical Game AI

- Finite state machine
  - Hard-coded behaviors, specify all possible combinations of actions
- A-star path planning
  - On top of a ~~waypoint graph~~ navigation mesh
  - Also local steering forces, object avoidance, etc.
- These techniques are widely used today

  ...But are not the focus of today's lecture

Alternative: Goal-Driven AI

- Hierarchical set of goals
  - e.g. "buy sword" => "obtain gold" + "go to smithy"
- Plan & perform actions to achieve goals
  - Planner finds a sequence of actions that will achieve a goal
  - Sequence is *not* hard-coded, discovered on the fly
- May also use A-star path planning

Planning Challenges

- Combinatorial explosion of world states
  - Need to simplify representation of world
  - Only model what is relevant to a specific agent
- Efficiently searching through possible actions
  - Can model as graph search problem
- Biasing towards certain actions
  - Weights on graph

Case Study: F.E.A.R

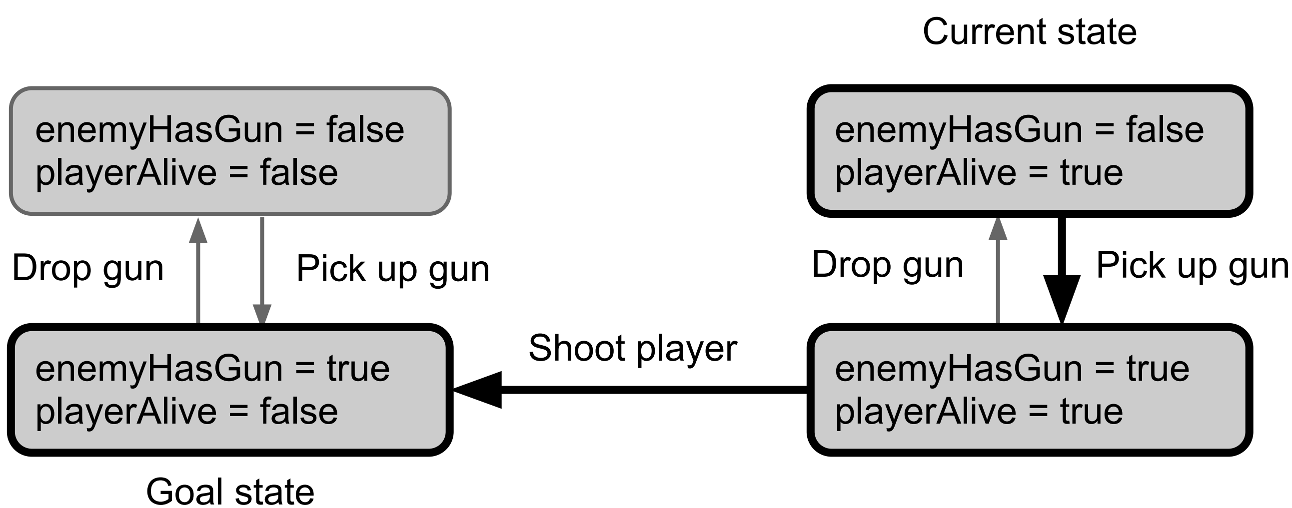
- 2005 horror FPS
- Ranked #2 Most influential AI Game

F.E.A.R. AI

- Uses GOAP (Goal-Oriented Action Planning)
  - Consists of goals and actions
  - Goal: certain state of the world we want to reach
  - Action: set of preconditions and effects
- Solved using graph search
  - Nodes are world states and edges are actions
  - Search from current state toward goal
  - Edges are directional (directed graph)
  - Analogy to chess AI: nodes are board states and edges are moves

GOAP Example

- World state:
  - bool enemyHasGun
  - bool playerAlive
- Only models states relevant to agent



GOAP

- Different from FSMs
  - FSM: Spend all time in nodes (actions), transitions are instant
  - GOAP: Spend all time in transitions (actions), nodes are instant
- Advantages over FSMs
  - Don't need to encode all possible transitions from each action to each other action
  - Simpler to specify and easier to scale
  - Just specify preconditions and effects for each action

GOAP in F.E.A.R.

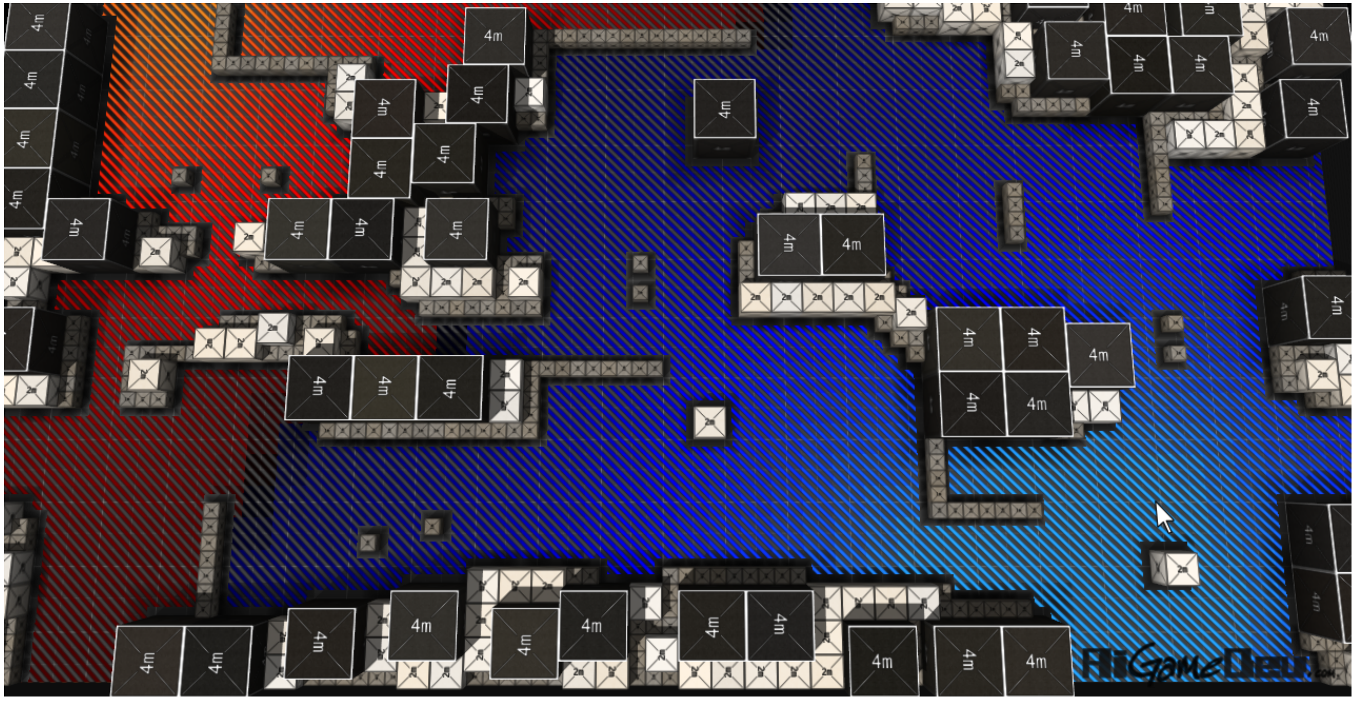
- Graph is dynamic
  - Edges are actions with preconditions and effects
  - Edges come and go based on game state
- Procedural preconditions
  - Taking an action may be currently impossible
  - Example: Escape through door requires door unlocked
- Procedural effects
  - Effects take time to execute and may fail
  - Example: Firing a projectile is blocked by something

GOAP in F.E.A.R.

- Don't want all agents to act the same
  - Vary edge weights based on individual preference (aggressive vs careful individual)
  - Need variety in available actions (run, crouch, dodge, roll, slide)
- Hard for player to understand AI's behavior
  - Add audio cues for player
  - "Send in reinforcements!"

Influence Maps

- Goal: decide which team controls an area
  - Used as an input to higher level decision algorithms



Influence Maps

- Details
  - Numeric value that varies throughout the world
  - Positive for team A, negative for team B
  - Absolute value indicates strength of influence
  - Can also encode other information: safety, congestion, etc...
- Advantages
  - Entire influence map not updated immediately, will "remember" recent history
  - Helps reason about strategy in complex worlds (open areas, choke points)
  - Commonly used in RTS games

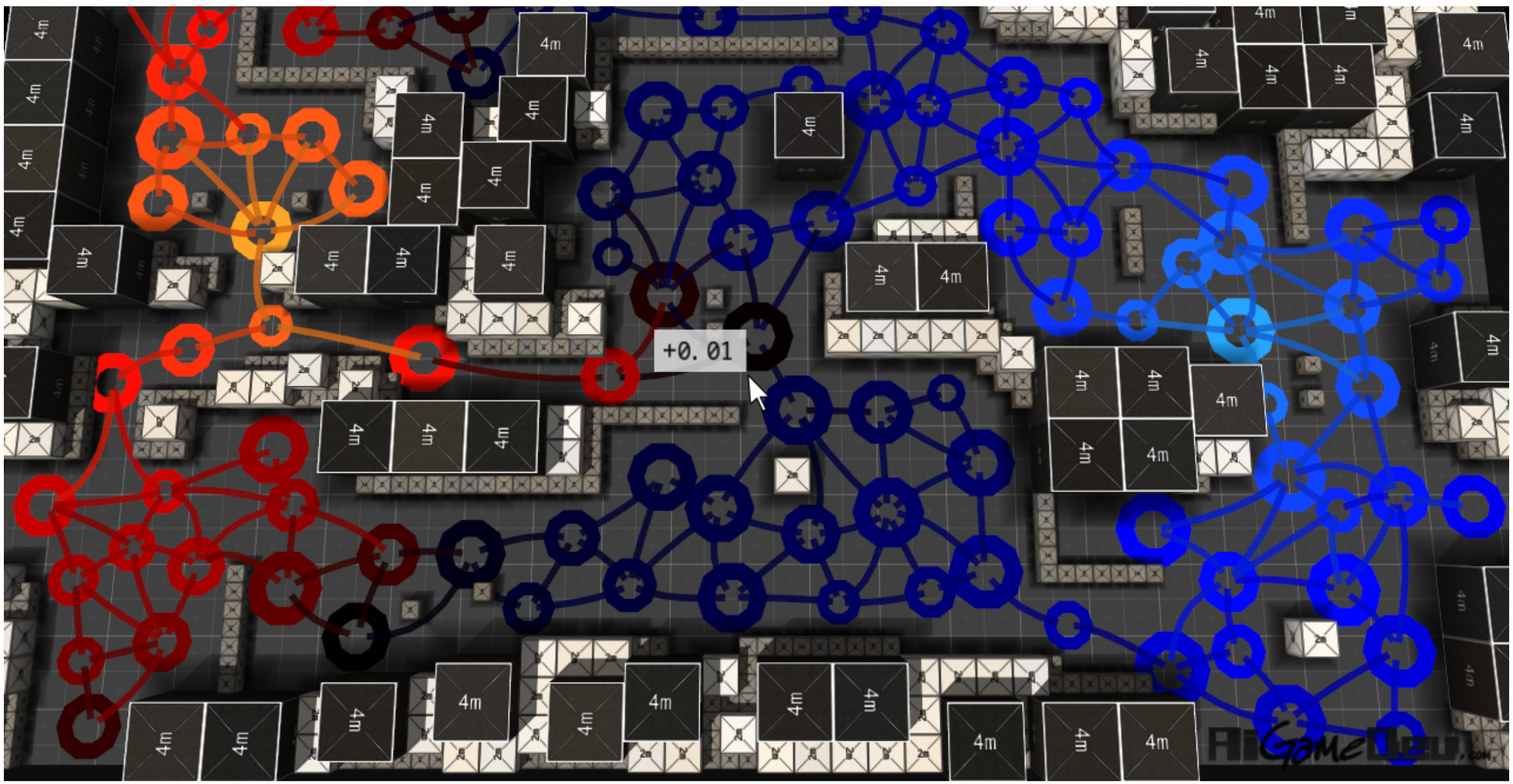
Influence Maps

- Algorithm
  - Influence map starts off at zero
  - Sources of influence are marked
    - Entities (friends and foes)
    - Events like grenade explosions, taking damage
  - Influence is propagated (diffused) to neighbors
- Parameters
  - Momentum (m): How fast do new values overwrite old values?
  - Decay (d): How fast do values decrease from the source?

$$x(t+1) = (1-m) \cdot x(t) + m \cdot \sum_{a \in agents} \left(x_a \cdot e^{-dist_a \cdot d} \right)$$

Influence Maps

- Influence maps can be combined with graphs



Neural Networks

Not used in commercial games

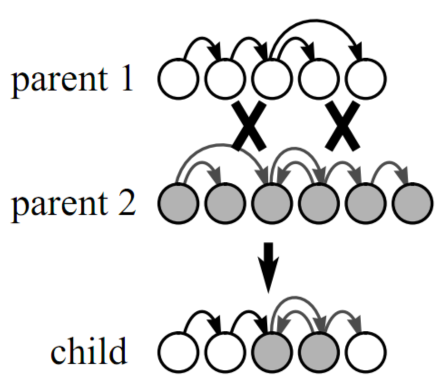
Evolved Virtual Creatures

- Develop a creature and a controller using a genetic algorithm
  - Research project by Karl Sims in 1994
- Start with randomized creatures
  - Configuration of body parts in a tree
  - Brain is a graph
    - Sensors (joint angle, contact, light)
    - Neurons (sum, sin, sigmoid, ...)
    - Effectors (a degree of freedom)
  - Goal defines fitness function
    - Example: jumping measures height of lowest body part

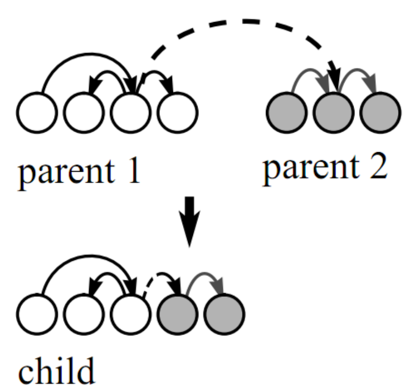
Evolved Virtual Creatures

- Evolve the current generation by randomly applying one of:
  - Mutation (asexual reproduction)
  - Crossovers (sexual reproduction)
  - Grafting (sexual reproduction)

Crossovers



Grafting



Evolved Virtual Creatures

http://www.youtube.com/watch?v=JBgG_VSP7f8

Genetic Algorithms in Games

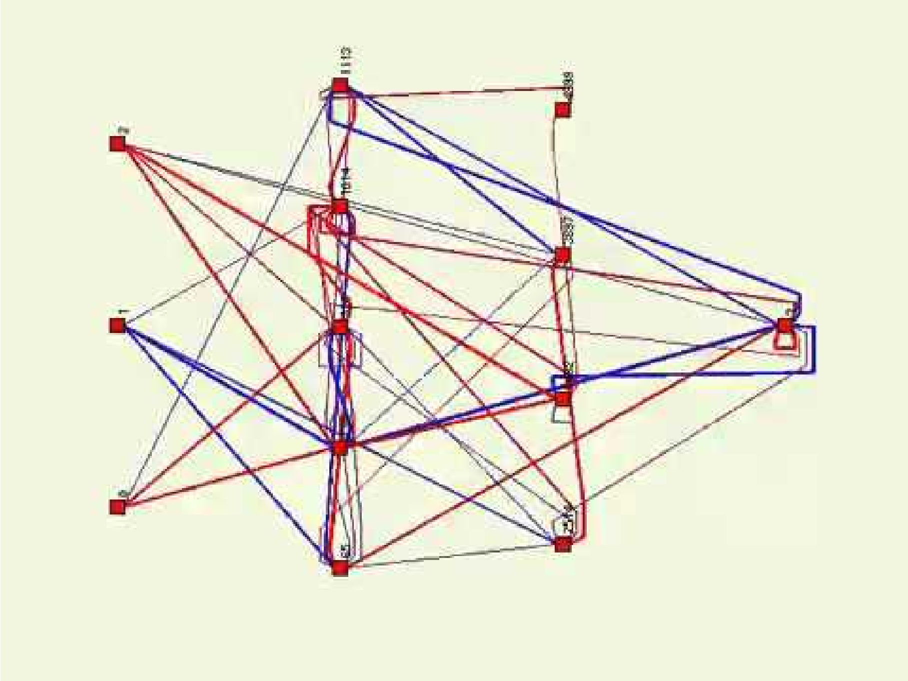
- Hardly ever used
  - Developers want fine control over behavior, often just enumerate all the cases
  - Difficult to test
  - Genetic algorithms take thousands of iterations to converge
- Creating faster genetic algorithms is an active area of research

Case Study: NERO

- Full name: Neuro-Evolving Robot Operatives
- Machine-learning based game
  - Started in 2003 as research project at UT Austin
- RTS-style gameplay with two phases of play
  - Phase 1: Train agents in sandbox scenarios
  - Phase 2: Simulate a battle between two groups of agents
- Works through a real-time genetic algorithm
  - NEAT (Neuro-Evaluation of Augmented Topologies)

NEAT Algorithm

- Neural network evolves over time
  - Evolution can change both connection weights and network structure



http://www.youtube.com/watch?v=T4EopjWkLtI

Real-time NEAT

- Start with minimally-connected neural network
  - Add connections that help solve the problem
- Continuously adapts a small group of agents
  - Doesn't use generations: this takes too long
  - Each "brain" has a evaluation time, when it runs out, it's replaced by merging two high-performance brains
    - Removed "brain" is shelved for later evaluation
- Open-source
  - http://nn.cs.utexas.edu/keyword?rtneat

Real-time NEAT in NERO

Every *n* game ticks:

1. Remove agent with worst adjusted fitness
2. Re-estimate fitness for all species
3. Choose a parent species
4. Adjust species boundaries and reassign all agents to species
5. Add new agent to the world

Case Study: Learning Bots in Quake

- Remco Bonse et. al. in 2004
- Use the Q-learning algorithm to train a neural network for an AI bot in Quake III
  - One-on-one game against preprogrammed AI
  - Replaced combat movement subsystem with NN
  - Reward is given for avoiding damage
- State:
  - Distance and angle of opponent
  - Distance and angle of nearest projectile
- Actions:
  - All 18 combinations of WASD + jump

Case Study: Learning Bots in Quake

- Q-learning algorithm:
  - Agent is in some state s ∈ S
  - Agent can take some action a ∈ A
  - State/action quality function: Q : S × A ⟶ ℝ
- Agent gets reward for each state change
  - Make a correction to Q based on new information
  - Learning rate: 0 = learn nothing, 1 = ignore past
  - Discount factor: 0 = short-term, 1 = long-term

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t(s_t, a_t)}_{\text{learning rate}} \times \left[\overbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_{a_{t+1}} Q(s_{t+1}, a_{t+1})}_{\text{max future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right]$$

Case Study: Learning Bots in Quake

- Conclusion
  - Best parameters were to play to 100,000 frags on a network of 15 neurons (10 hours training time)
  - Learned bot wins twice as often against a preprogrammed bot
  - Human players didn't notice a difference
  - Limited interaction time in a FPS

Randomness and AI

- One motivation for AI is variability
- Adding randomness can help
  - Easy way of varying AI behaviors
  - Need to be careful about how user's perceive randomness

Randomness and Human Perception

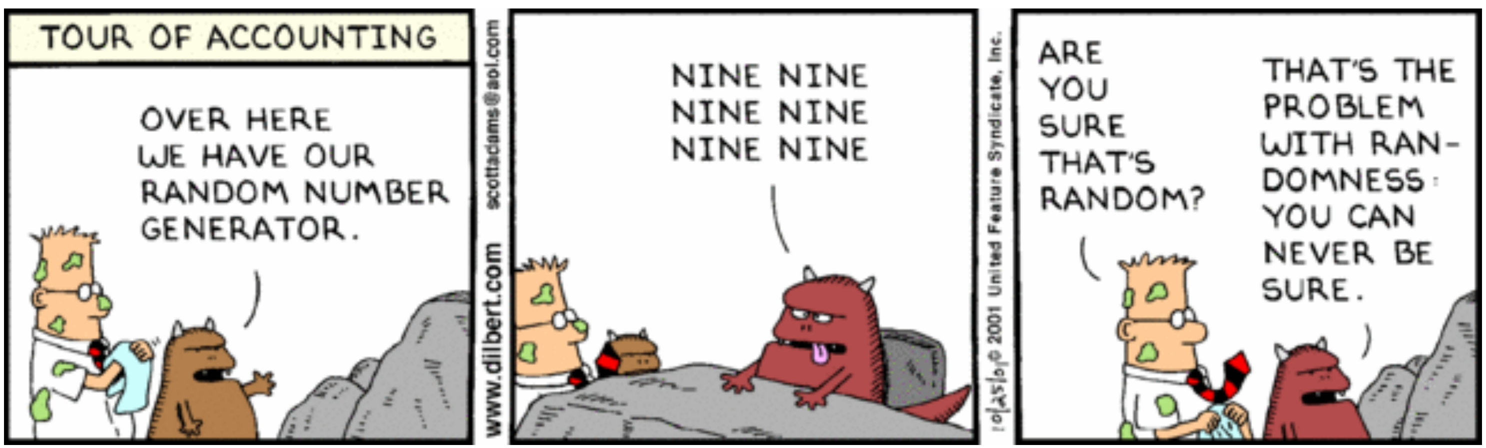
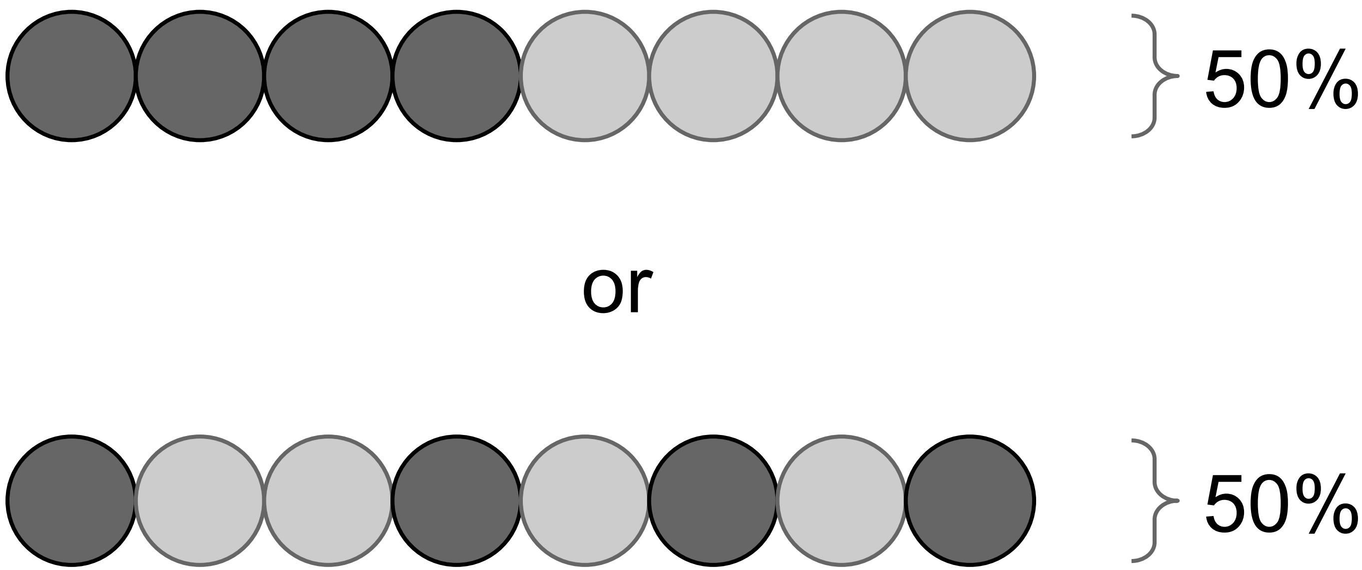


Image from dilbert.com

Randomness and Human Perception

- Which is more random?

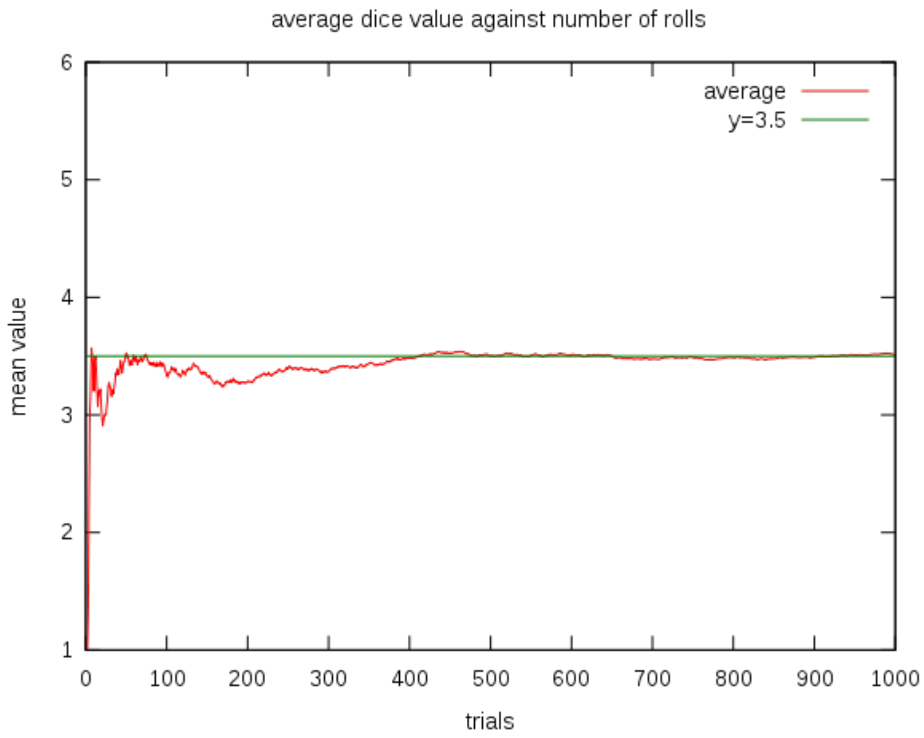


Randomness and Human Perception

- Humans are illogical
  - Conditioned to see patterns
  - Patterns are more memorable
  - "He got 3 but I only got 1"
- Gambler's Fallacy
  - We think outcomes depend on past events for independent events with fixed probabilities
  - Not just gamblers, everyone does this
  - "His luck will run out sooner or later"

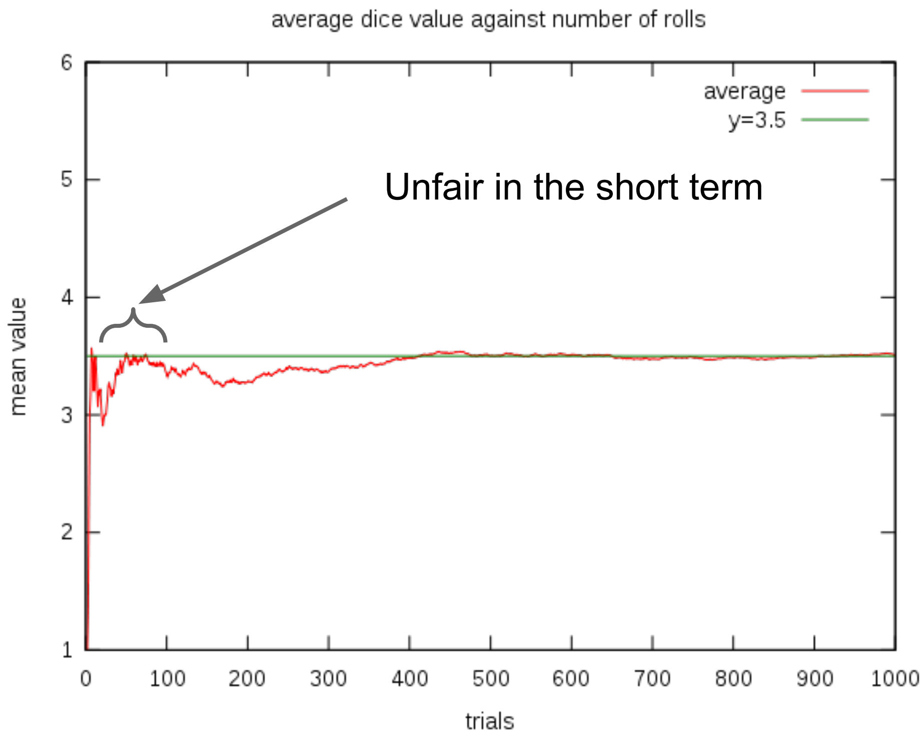
Law of Large Numbers

- Average approaches the expected value as the number of trials increases



Law of Large Numbers

- Average approaches the expected value as the number of trials increases



Law of Small Numbers?

- Make randomness easier to understand
  - Random choices without replacement
  - Keep track of past choices and prevent them from being chosen again too soon
  - Humans expect random binary choices will alternate up to 40% more than is mathematically reasonable[1]
  - Example: If a player loses a game involving randomness 4 times in a row, is that a good experience?

[1] A Meta-Analysis of Randomness in Human Behavioral Research

C++ Tip of the Week

- Multiple inheritance
  - C++ classes can have multiple superclasses
  - Can be used to emulate Java interfaces

```
// Declare interfaces using pure virtual functions
struct Reader { virtual char read() = 0; };
struct Writer { virtual void write(char) = 0; };

// File * can be cast to Reader * and Writer *
struct File : Reader, Writer {
  char read() { ... }
  void write(char) { ... }
};
```

C++ Tip of the Week

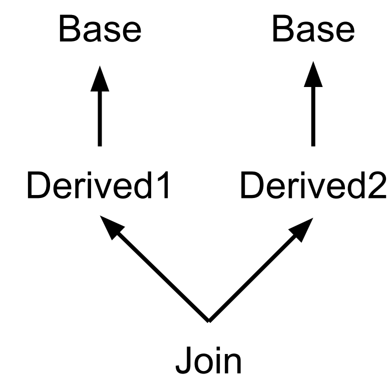
- More powerful than Java interfaces
  - All superclasses can have state

```
struct Timestamp {
  const int creationTime;
  Timestamp() : creationTime(time(0)) {}
};
struct UniqueID {
  const int id; static int nextID;
  UniqueID() : id(nextID++) {}
};

struct Entity : Timestamp, UniqueID {};
```

C++ Tip of the Week

- The diamond problem and virtual inheritance

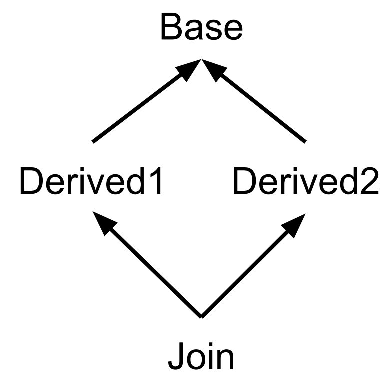
```
struct Base { int num; };
struct Derived1 : Base {};
struct Derived2 : Base {};
struct Join : Derived1, Derived2 {};

Join join;
cout << join.num; // error, join has two copies of num
cout << join.Derived1::num;
cout << join.Derived2::num;
```



C++ Tip of the Week

- The diamond problem and virtual inheritance

```
struct Base { int num; };
struct Derived1 : virtual Base {};
struct Derived2 : virtual Base {};
struct Join : Derived1, Derived2 {};

Join join;
cout << join.num; // works, join has one copy of num
```



C++ Tip of the Week

- Tricky case
  - Multiple pointers for join (base1 != base2)
  - Using virtual inheritance here would cause an error

```
struct Base { virtual void foo() = 0; };
struct Derived1 : Base { void foo() { cout << 1; } };
struct Derived2 : Base { void foo() { cout << 2; } };
struct Join : Derived1, Derived2 {};

Join join;
Base *base = &join; // error, ambiguous conversion
Base *base1 = static_cast<Derived1 *>(&join);
Base *base2 = static_cast<Derived2 *>(&join);
```

C++ Tip of the Week

- Tricky case
  - Fix: define the behavior we want by overriding foo() again in the class with multiple inheritance

```
struct Base { virtual void foo() = 0; };
struct Derived1 : virtual Base { void foo() { ... } };
struct Derived2 : virtual Base { void foo() { ... } };
struct Join : Derived1, Derived2 {
  void foo() { Derived1::foo(); Derived2::foo(); }
};

Join join;
Base *base = &join; // works, only one foo() implementation
```

C++ Tip of the Week

- Common use case: policy-based designs
  - Compile-time version of the strategy pattern
  - Each template parameter provides part of behavior
  - Compiler can inline and optimize policy methods

```
template <typename K, typename V, class Alloc, class Hash>
struct HashMap : Alloc, Hash {
  V &get(const K &key) {
    Bin &bin = bins[hash(key)];
    if (!bin.contains(key)) bin.insert(key, alloc());
    return bin.find(key);
  }
};
HashMap<Point, Entity, MemoryPool, SecureHash> data;
```

C++ Tip of the Week

- C++ mixins
  - Alternative to multiple inheritance
  - Inherit from a template parameter
  - Use initializer functions instead of constructors if you need arguments on construction

```
template <typename Base>
struct Timestamp : Base { ... };

template <typename Base>
struct UniqueID : Base { ... };

Timestamp<UniqueID<Entity>> entity;
```

Final Project Questions?

References

- Using Randomness in AI: Both Sides of the Coin (Dave Mark and Brian Schwab)
- http://web.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf
- Evolving Virtual Creatures (Karl Sims)
- Evolving Neural Network Agents in the NERO Video Game (Kenneth Stanley et al.)
- A Meta-Analysis of Randomness in Human Behavioral Research (Summer Ann Armstrong)
- Learning Agents in Quake III