Advanced Gameplay Programming

Week 3

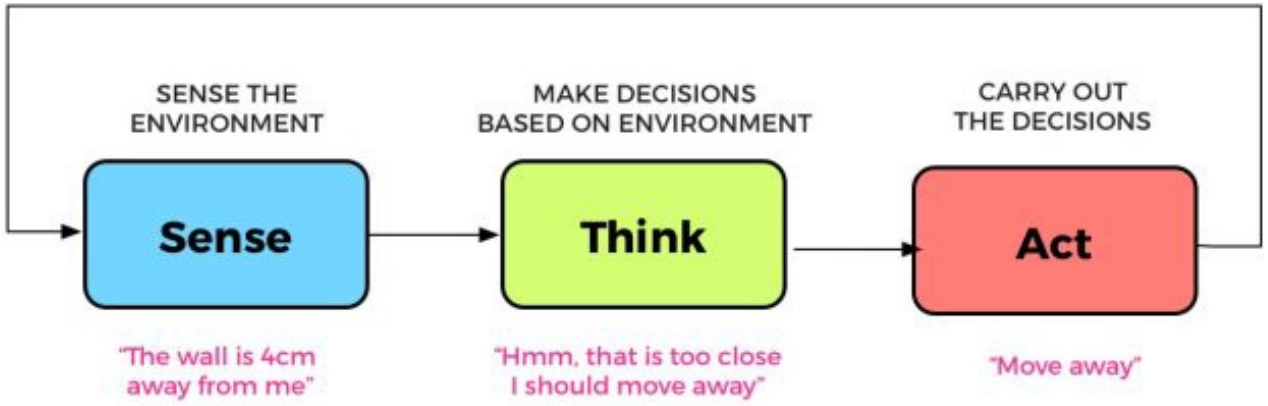
HW Review

Smarter Play

AI Decisionmaking

What is Game AI?

- What actions a game entity should take, based on current conditions.
- “Sense / Think / Act” cycle



Constraints of Game AI

- Difficult to do machine learning
  - No players to observe
  - The game is constantly changing
- Games shouldn’t be optimal
- Want to model “realistic” or human-like behavior
- Must be real-time, while the rest of the game is running
  - Can’t monopolize all the system resources
- System should be data-driven, rather than hard coded
  - Want non-technical designers to be able to adjust, and the solution to change as the game is developed

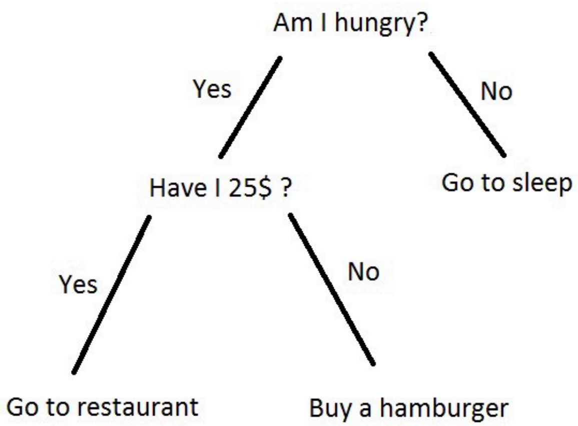
Hard-Coded Conditionals

- Simple, reactive AI
- PROS:
  - Simple to write
  - Easy to update (while there are few conditions)
  - Easy for non-technical designers to read and understand.
  - Almost no “sense” or “think” – mostly ”act”
- CONS:
  - You tell me!

```
private void Update()
{
    if (BallIsLeft())
    {
        MoveLeft();
    }
    if (BallIsRight())
    {
        MoveRight();
    }
}
```

Decision Tree

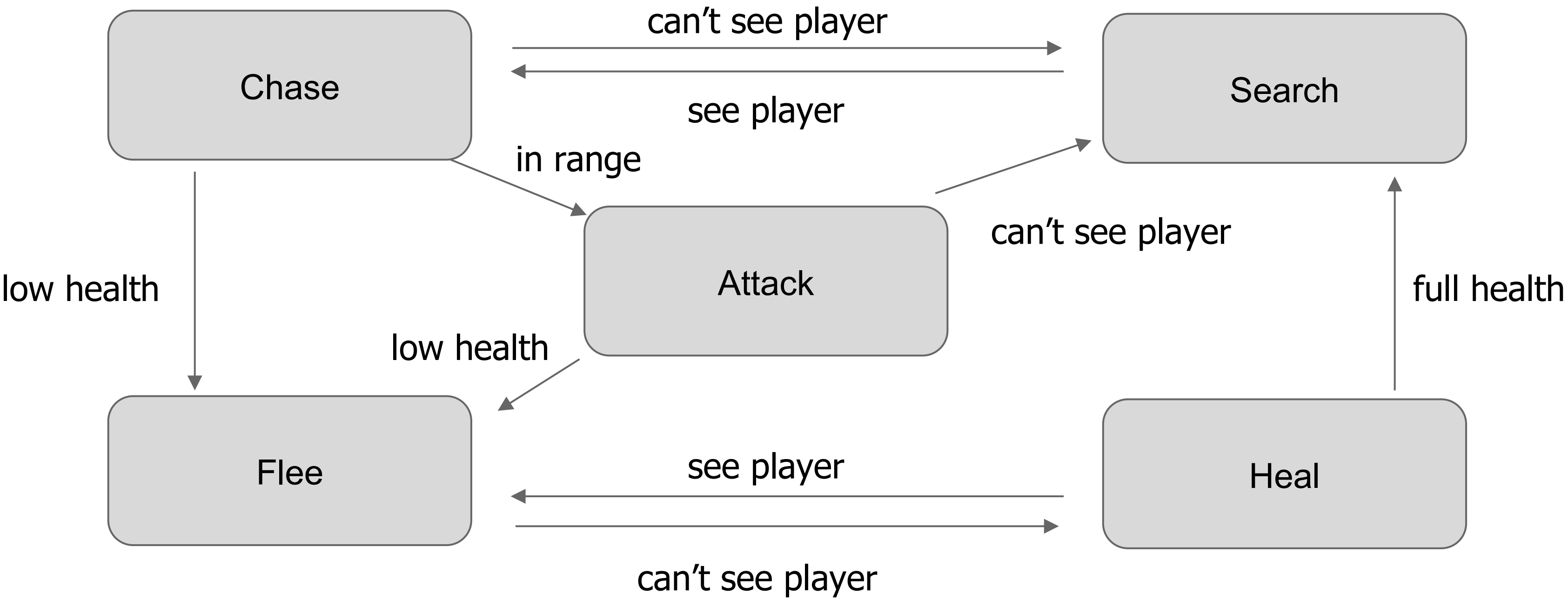
- Represent decision as series of edges
- Leaf nodes are "actions"
- Functionally, identical to dumb, reactive AI
- Logically easier to read.
- PROS:
  - More readable for more complicated logic
  - Scales slightly better than Hard-Coded Conditionals
- CONS:
  - You tell me!



Event Reactive

- The game controller shoots events for specific actions
- Imagine “wait to start partying until door is opened”
- PROS:
  - Reduces use of game resources
  - Can create very complicated behavior
- CONS:
  - You tell me!

Finite State Machine



Finite State Machine

- AI contains a finite state machine, and can go between different states
- PROS:
  - Reduces use of game resources (in some ways)
  - Can create very complicated behavior
- CONS:
  - You tell me!

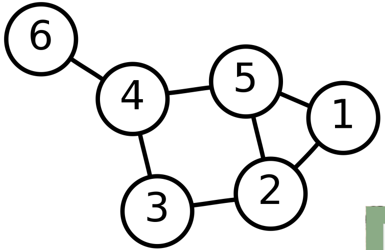
Hierarchical State Machine

- States contain their own state machine
- Can create a “non-combat” state that has more complicated behavior
- Create a subclass of State called “SubState”
    - Takes a state as it’s context
    - Each State has a finite state machine that runs it’s sub-states
- PROS:
    - Can model more complicated behavior
    - Can organize more complicated behavior
- CONS:
    - You tell me!

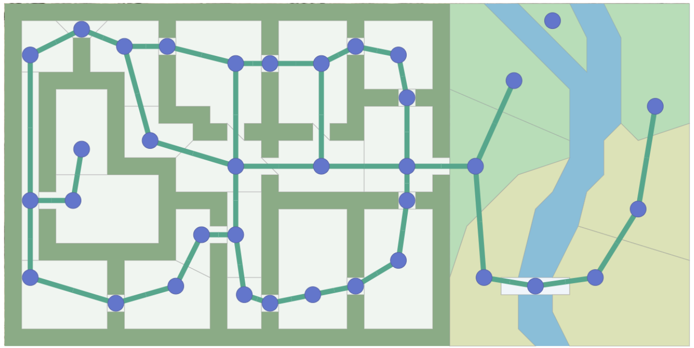
What are the main issues?

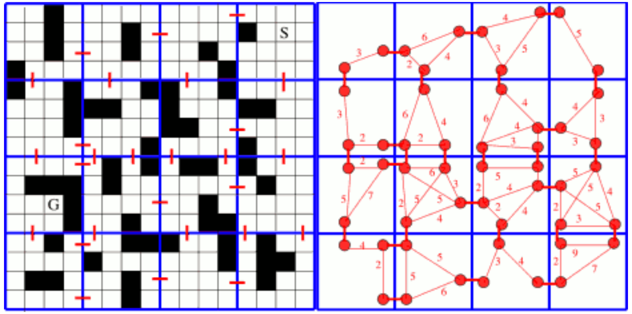
- Not reusable (except for state machines, sometimes)
  - All the hard-coded conditions are difficult to make generic
  - Hard-coded transitions between states mean all states have to be present
- Not scalable
  - Changing one state means updating a lot of other states
  - Adding conditions means updating almost all the logic
- Not accessible to non-technical designers
  - Hard to make visual representation (look at Unity’s Animator Window)
  - AI is gameplay, so it’ll need to be adjusted often
- Sometimes you want transition rules that apply no matter which state you’re in, or which apply in almost all states.

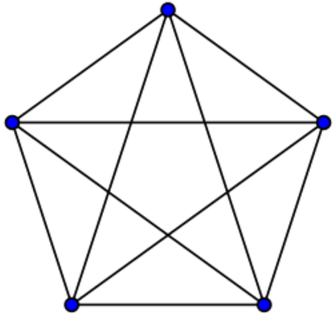
Trees / Graphs



- What is a graph?
  - Nodes
  - Vertices
    - Directed
    - Undirected
- Can have a lot of properties
  - Weighted vertices
  - Connected
  - Cycles / Acyclic
  - Complete
  - “Tree”
- Useful:
  - Searching
  - Pathfinding
  - Organizing data



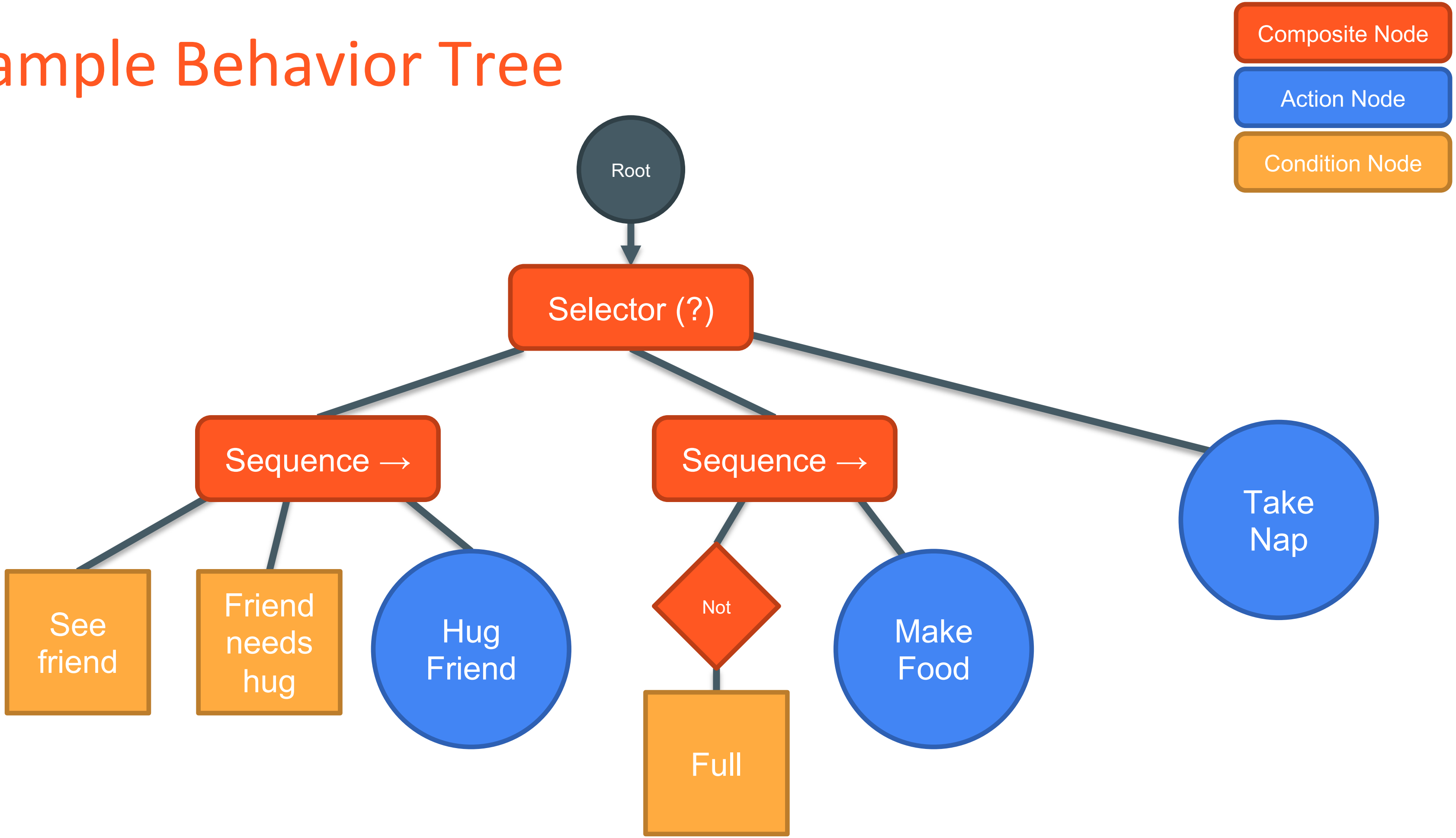




Behavior Trees

- Originally developed by Bungie for Halo 2
- Behavior trees are “trees”
  - Graphs made up of nodes and vertices
- Finite State Machines are “heavy graphs”
  - Logic and behavior in same node.
- Behavior trees are lighter
  - Separate logic and behavior into separate nodes

Example Behavior Tree



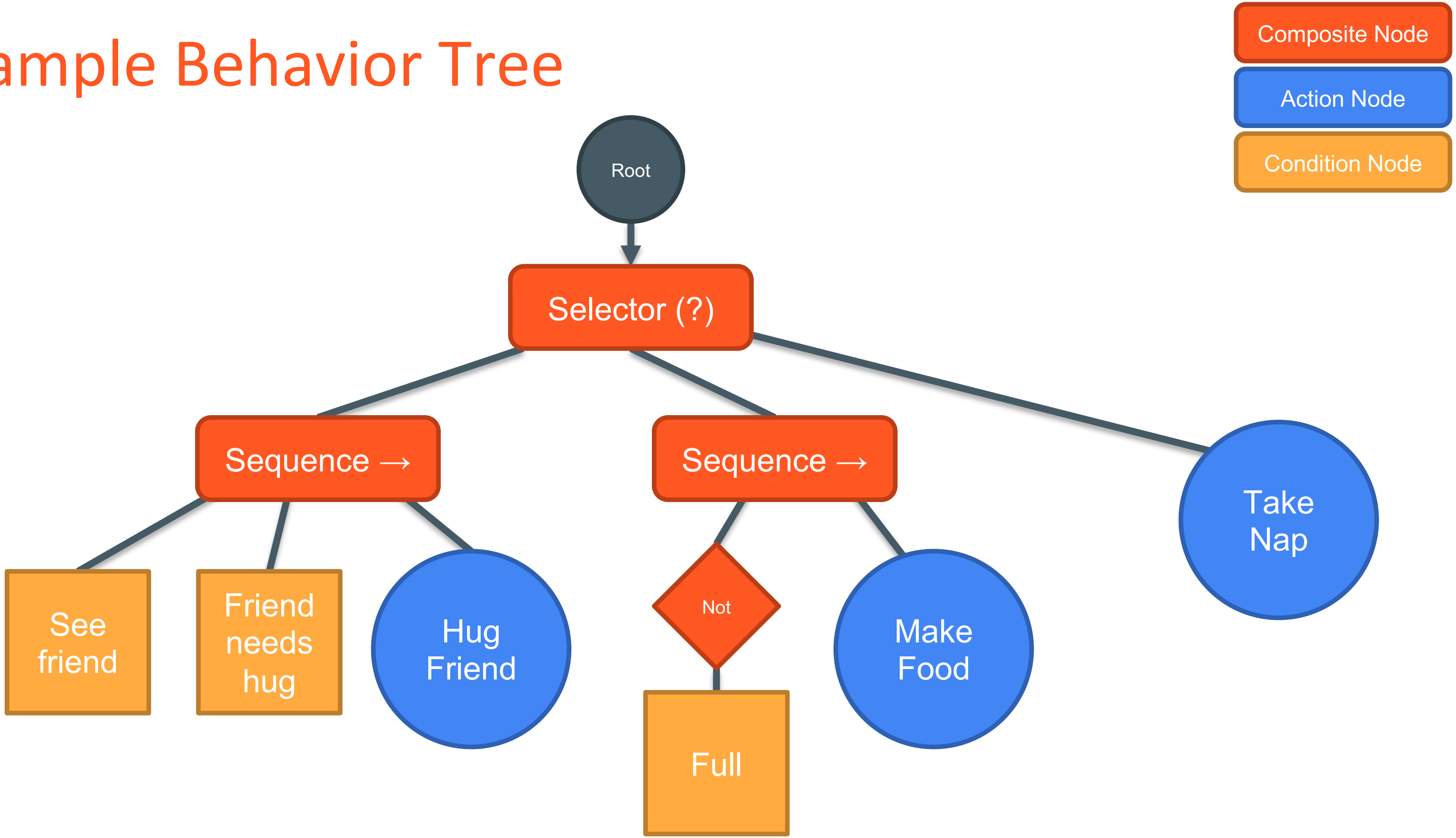
Behavior Tree Nodes: Composite

- *Selector* (?):
  - Select one of their children.
  - Return success as soon as **one** of their children is successful
  - Return failure when **all** children fail
- *Sequence* (→):
  - Act like checklists.
  - Return success when **all** their children are successful
  - Return failure as soon as **one** of their children fails
- *Decorator* (◇):
  - Have a single child and modify the return value of that node
  - Inverter (return opposite of child)
  - Succeeder (always return true)
  - Repeater (run child X number of times)
  - Repeat until fail (run child until child fails)

Behavior Tree Nodes: Action

- Action Node(☐):
  - Nodes at the outer edges of the tree (i.e. the 'leaves’)
  - Usually means stop updating the tree and *do the action.*
  - Move towards chest, pick up object, sit down, etc.
- *Condition Node*(…):
  - Nodes at the outer edges of the tree (i.e. the 'leaves’)
  - Don’t create behavior or stop updating the tree
  - Used by composite nodes to decide where in tree to go next.

Example Behavior Tree



Let’s Go To The Code!

https://github.com/jackschlesinger/AGP_BehaviorTreeExample

Utility Based System

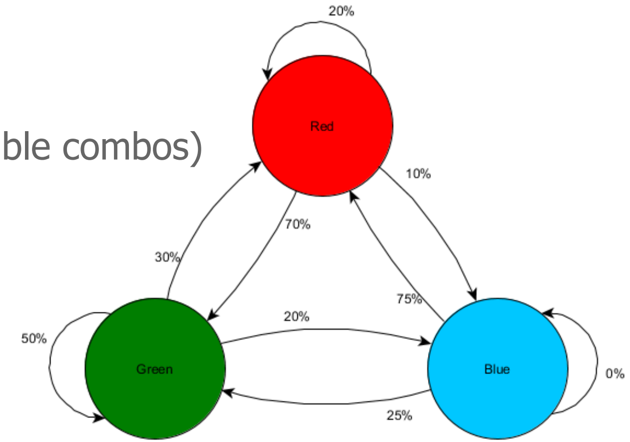
- Create a system of possible actions
- Assign function to determine value of each action
- Do highest “utility” action
- PROS:
    - Extremely useful for games with extremely discrete actions, and easily determined utiltity
    - A game of chess would be a good example.
    - Can create very human simulations
        - In “The Sims”, for example, each agent has “drives” and “motivations” that influence scores
    - Can be used with other systems:
        - behaviour tree could have a “Utility” composite node that uses utility scores to decide which child to execute.
- CONS:
    - By default, any state can follow any other state

Adaption: Weight Based

- Keep set values that inform decision making
- If a strategy is successful, weight that strategy higher in future play.
- If a strategy is unsuccessful, weight that strategy lower in future play.
- Imagine creating enemy AI for a FPS:
  - At first, picks a room at random
  - If killed in a room, decrease the weight of that room
  - If scores a lot of points in a room, increase the weight of that room.
  - AI avoids deadly rooms, and pursues successful rooms.

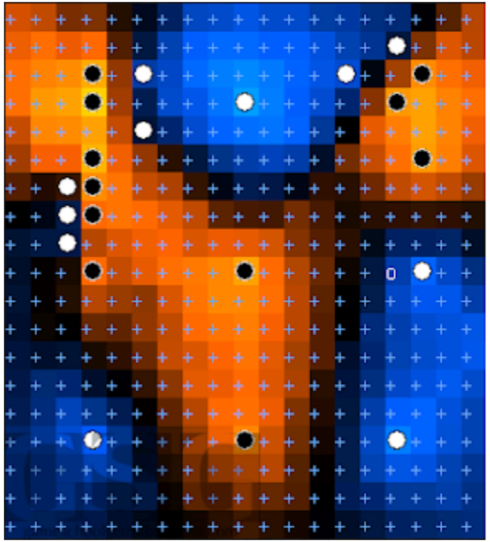
Adaption: Markov Model

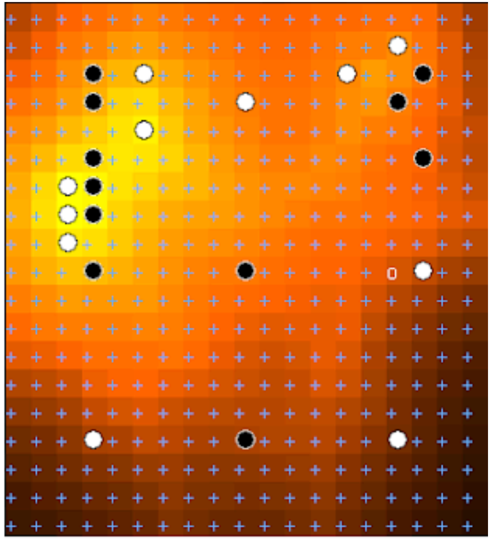
- Use knowledge of past states to predict future states
- Represented as a FSM with vertices w/probability of transition
- Can also predict the likelihood of a *series* of events
  - This is called a Markov Chain
- N-grams are an example of a simple Markov Model
  - Given N inputs, what's the likely next input?
  - Keep table of all inputs, and train assumptions
  - Doesn't scale well (for K possible inputs, have K^N possible combos)



Influence Maps

- Have each entity write out their “zone of influence”, and have that weight pathfinding and decisionmaking
- Very useful if your game can be divided into factions





Other topics

- Line of sight chasing
  - Pixel line drawing
  - Intercepting
- Blackboards
- Pattern Movement
- Flocking
- Fuzzy Logic
- Machine Learning
  - Deep Learning
  - Genetic algorithms
  - Neural Networks
  - Learning Element | Performance Element | Problem Generator | Critic

Advanced C# Topics

Advanced C# Topics

- Delegates
  - Action/ Func
  - User Defined
- Callbacks
- Lambdas
- Closures
- Reflection
- LINQ
- Interfaces

Advanced Unity Topics

- Scriptable Objects

Strategy Pattern

```
delegate void Attack(Enemy e);
class Player
{
    private Attack _attack;

    public Player(Attack attack)
    {
        _attack = attack;
    }

    public void Update()
    {
        var e = GetNearestEnemy();
        if (e !== null) _attack(e);
    }
}

// Somewhere else...
Attack attack = ChooseCustomAttackBasedOnRuntimeFunction();
Player player = new Player(attack);
```

References:

- A* Pathfinding: https://www.redblobgames.com/pathfinding/a-star/introduction.html
- Gameplay Programming Patterns: https://gameprogrammingpatterns.com/contents.html
- Mattia Romeo’s Programming Patterns Website: https://gpp.ghirigoro.net/
- Influence Maps: http://gameschoolgems.blogspot.com/2009/12/influence-maps-i.html
- Overview of Game AI: https://www.gamedev.net/articles/programming/artificial-intelligence/the-total-beginners-guide-to-game-ai-r4942/