CPSC 427
Video Game Programming

Debugging and Simulation

Helge Rhodin

Overview

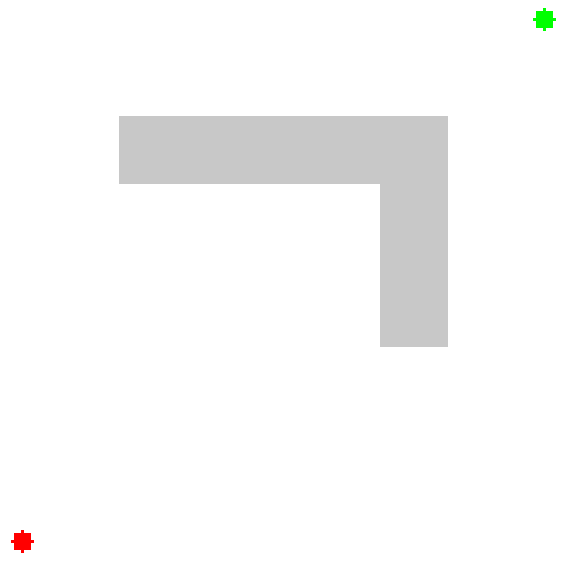
1. ***Recap AI***

2. ***Debugging***

3. ***Simulation***

Lowest-Cost-First Search (LCFS)

- **Lowest-cost-first search** finds the path with the **lowest cost** to a goal node
- At each stage, it **selects** the path with the **lowest cost** on the frontier.
- The **frontier** is implemented as a priority queue ordered by path cost.



Use of search

- Use search to determine next state (next state on shortest path to goal/best outcome)
- Measures:
  - *Evaluate goal/best outcome*
  - *Evaluate distance (shortest path in what metric?)*

Problems:

- Cost of full search (at every step) can be prohibitive
- Search in adversarial environment
  - *Player will try to outsmart you*

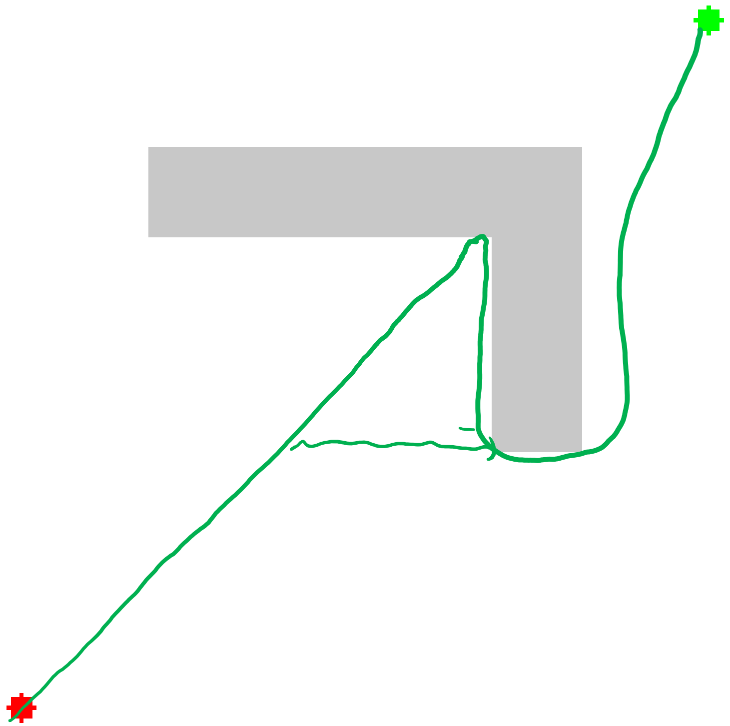
Heuristic Search

- Blind search algorithms do not take goal into account until they reach it
- We often have estimates of distance/cost from node n to a goal node
- **Estimate = search heuristic**
  - **a scoring function h(x)**

Best First Search (BestFS)

- Best First: always choose the path on the frontier with the smallest h value
  - *Frontier = priority queue ordered by h*
  - *Once reach goal can discard most unexplored paths…*
    - Why?
  - *Worst case: still explore all/most space*
  - *Best case: very efficient*
- **Greedy**: (only) expand path whose last node seems closest to the goal
  - *Get solution that is **locally** best*

A* search

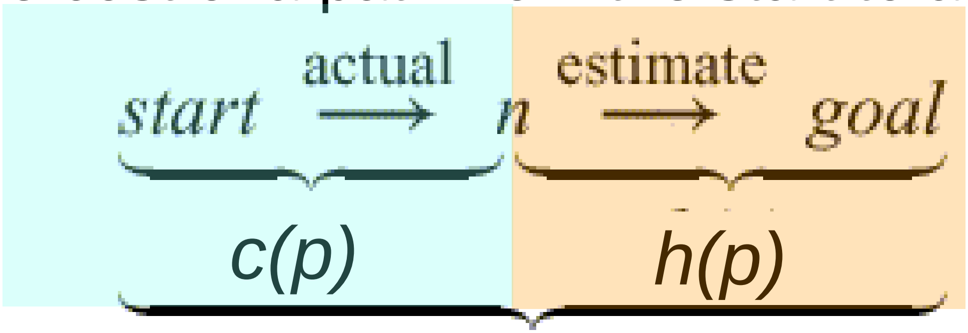


https://en.wikipedia.org/wiki/A*_search_algorithm

A* Search

- A* search takes into account both
  - *c(p)* = cost of path *p* to current node
  - *h(p)* = heuristic value at node *p (estimated "remaining" path cost)*

- Let *f(p) = c(p) + h(p).*
  - *f(p)* is an estimate of the cost of a path from the start to a goal via *p*.



A* always chooses the path on the frontier with the lowest estimated distance from the start to a goal node constrained to go via that path.

A* search

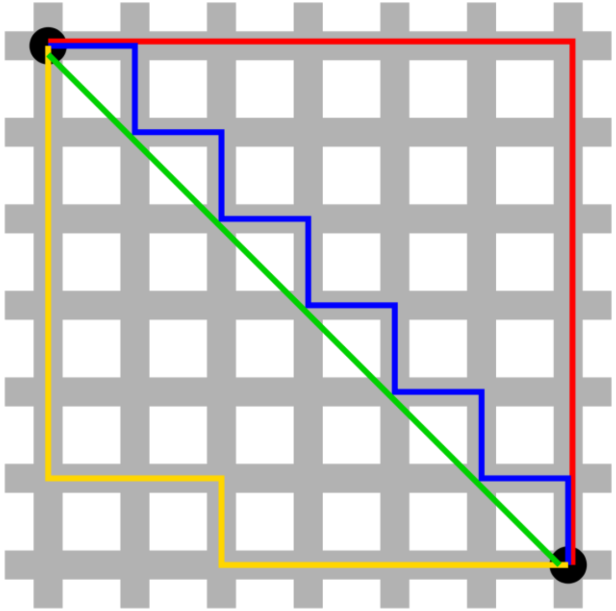
Key idea: H is a heuristic, and not the real distance:

h(p,q) = |(p.x – q.x)| + |(p.y – q.y)|

- Manhattan distance

h(p,q) = sqrt((p.x – q.x)^2 + (p.y – q.y)^2)

- Euclidean distance



https://en.wikipedia.org/wiki/Taxicab_geometry

Conditions:

- a heuristic function is **admissible** if it never overestimates the cost of reaching the goal
- a heuristic function is said to be **consistent**, or **monotone**, if its estimate is always less than or equal to the estimated distance from any neighbouring vertex to the goal, plus the cost of reaching that neighbour

Two-player games

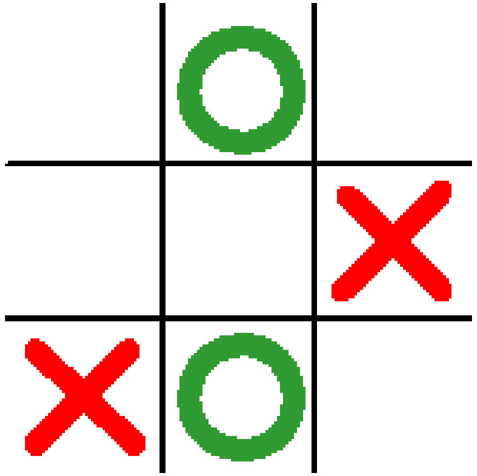
www.npr.org

Min-Max Trees

- Adversarial planning in a turn-taking environment
  - *Algorithm seeks to maximize our success* ***F***
  - *Adversary seeks to minimize* ***F***
- Key idea: at each step algorithm selects move that minimizes highest (estimated) value of F adversary can reach
  - *Assume the opponent does what looks best*

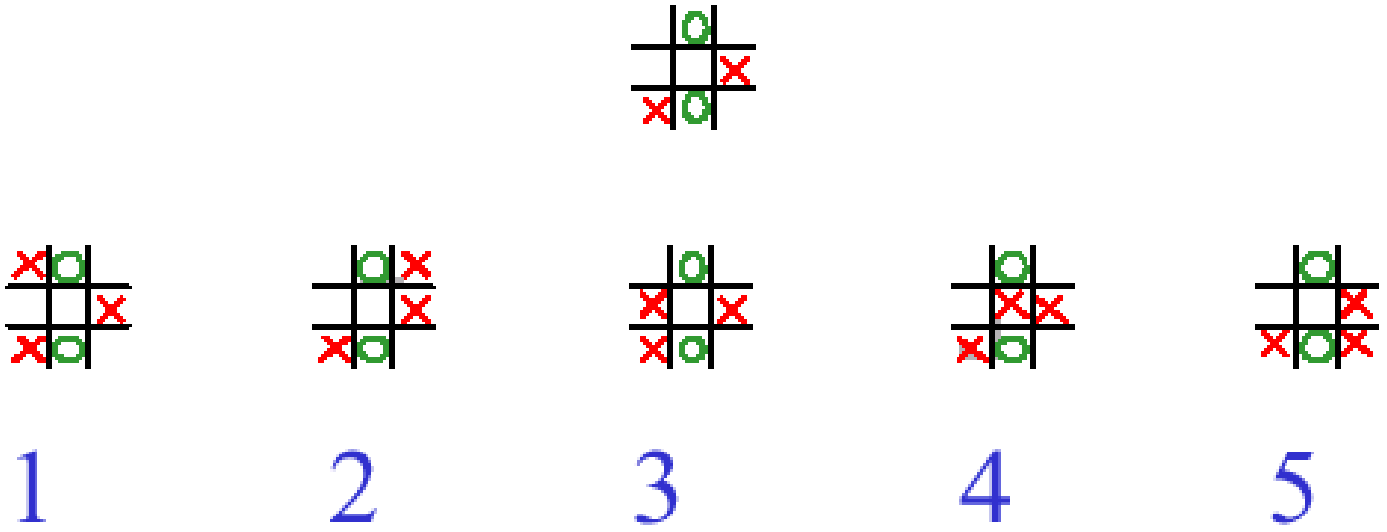
Example

(from *uliana.lecturer.pens.ac.id/Kecerdasan%20Buatan/ppt/Game%20Playing/gametrees.ppt*)



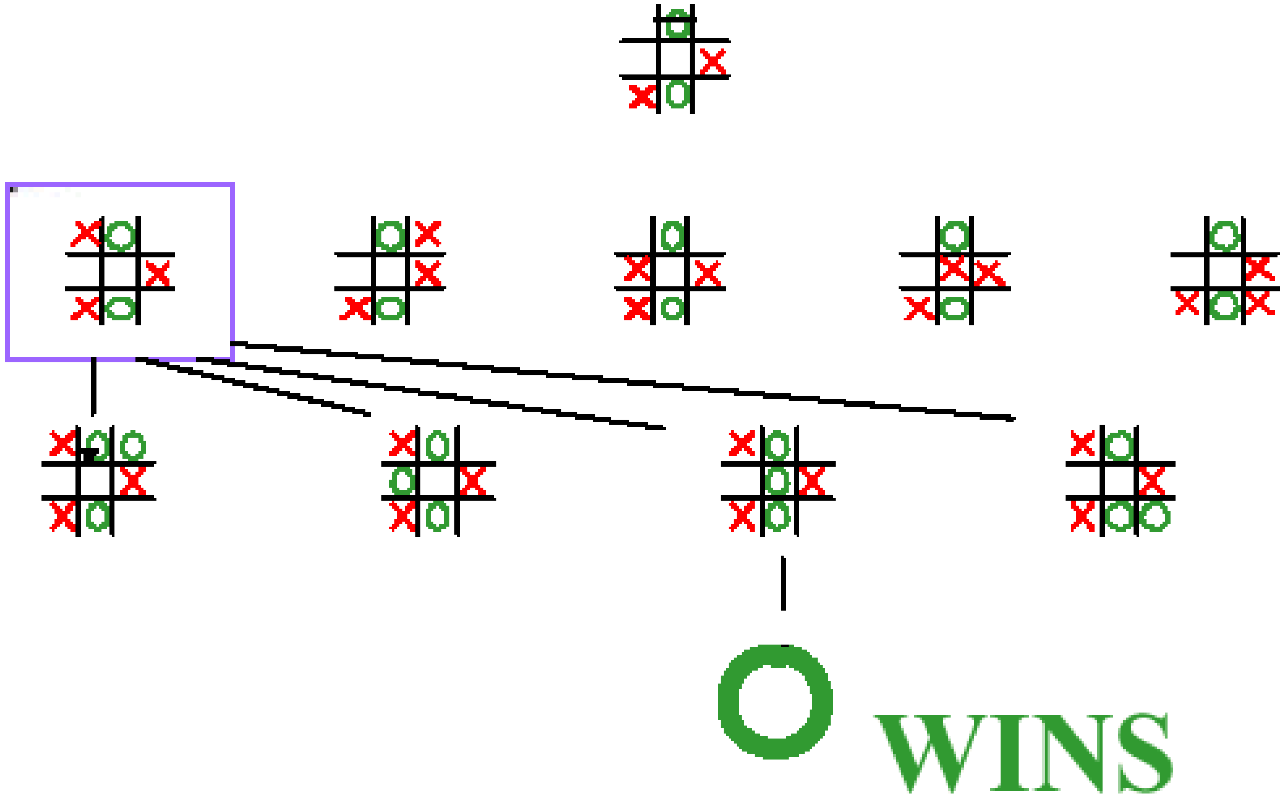
We are playing X, and it is now our turn.

Our options:



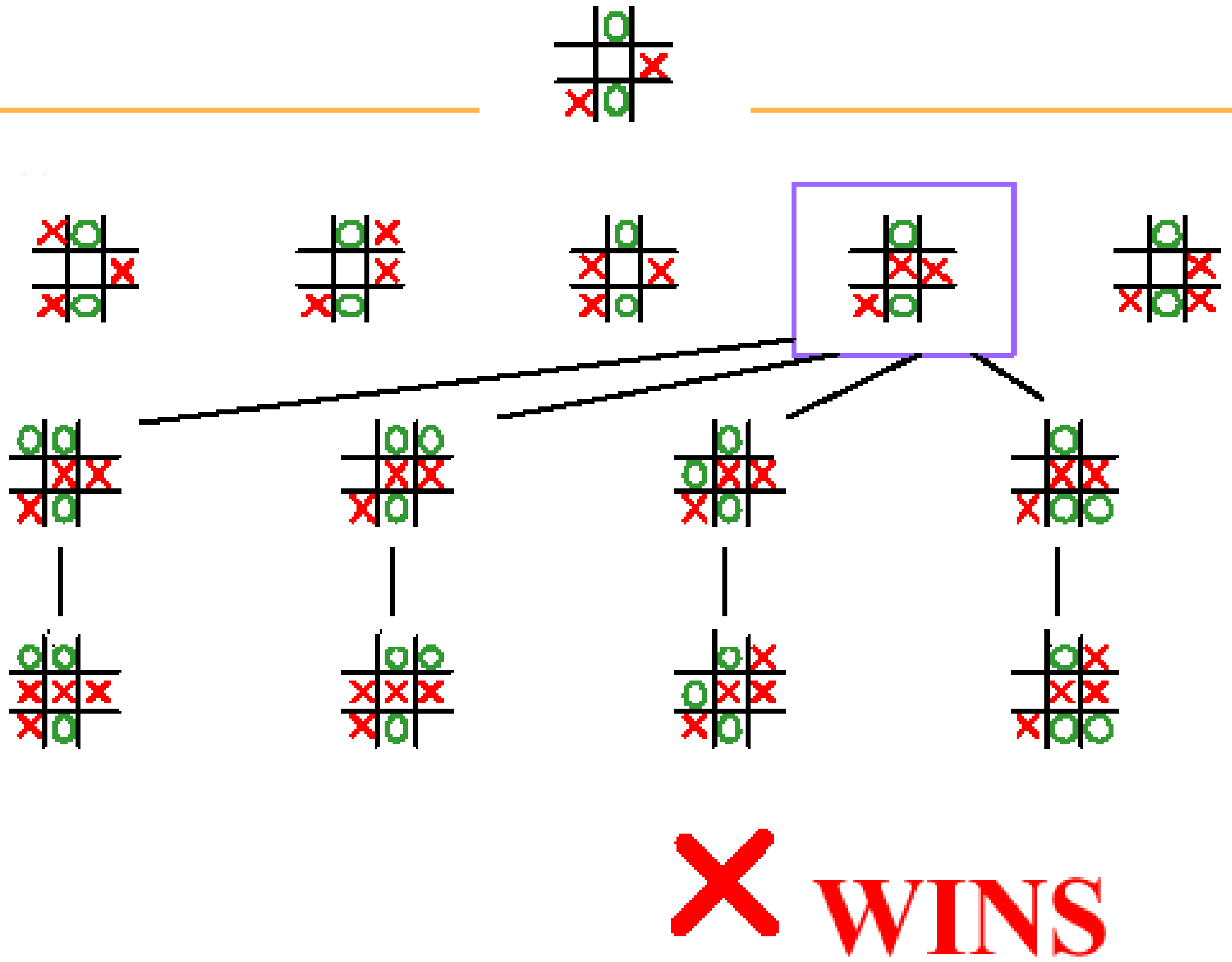
Number = position after each legal move

Opponent options



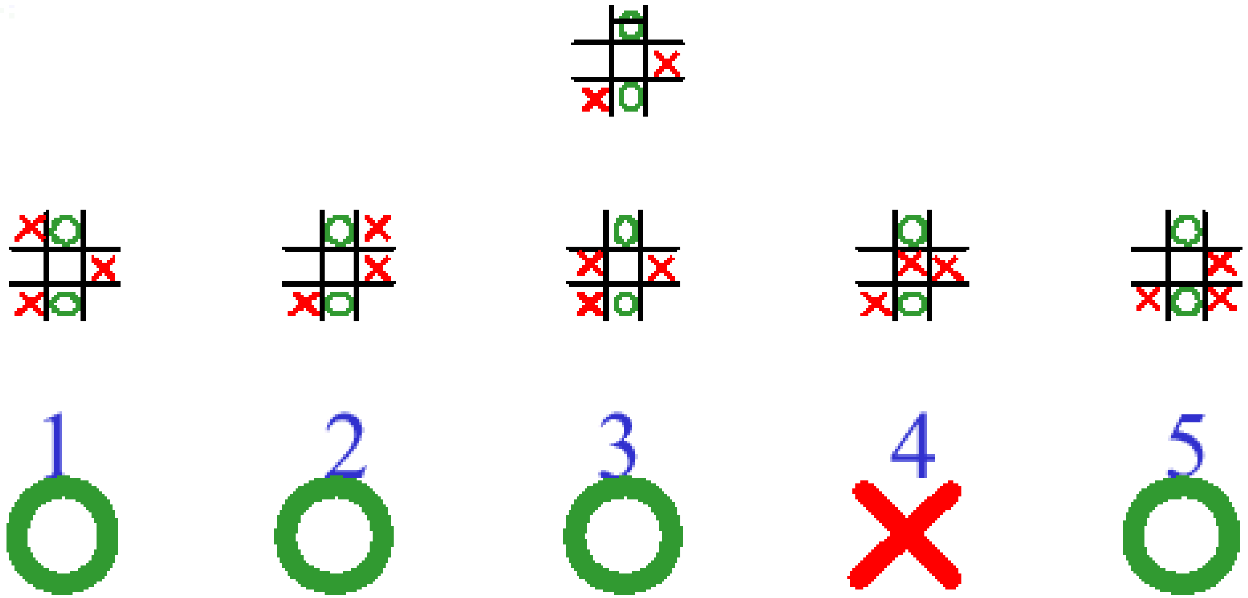
Here we are looking at all of the opponent responses to the first possible move we could make.

Our options



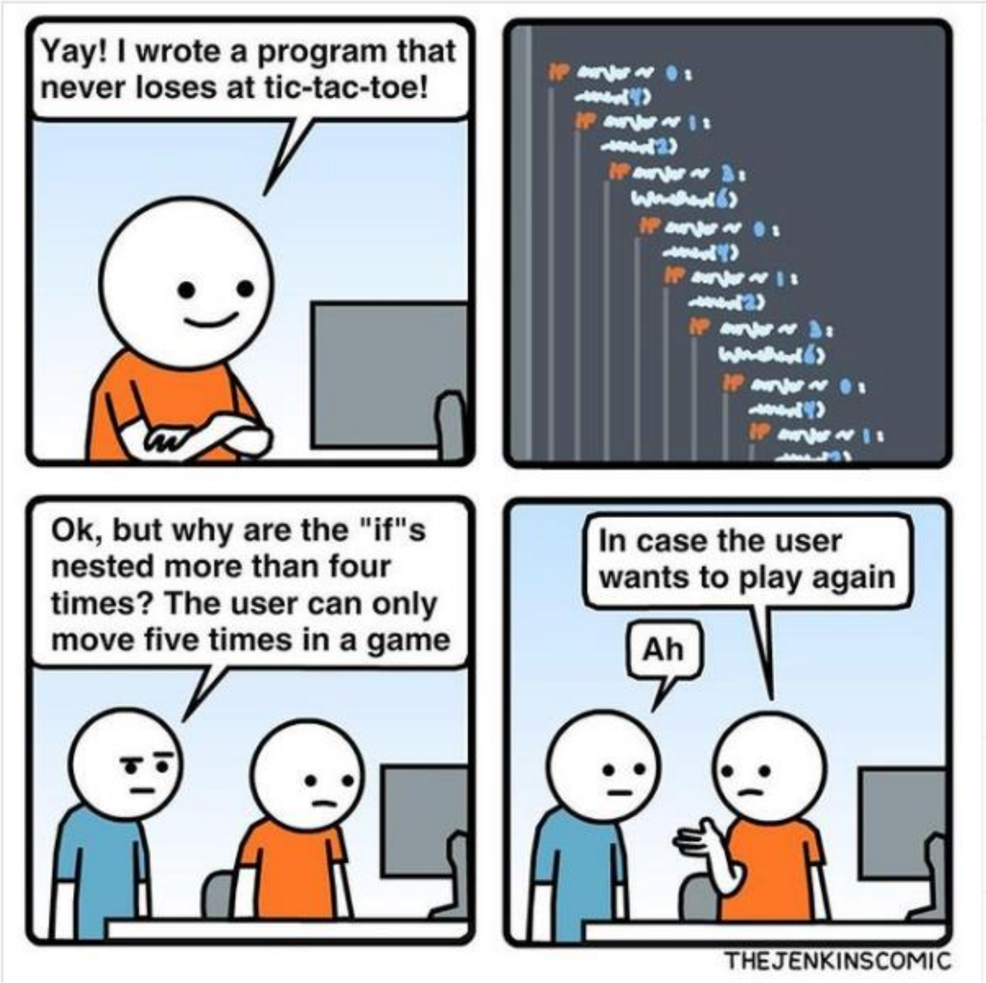
We have a win for any move they make.
Original position in purple is an X win.

Summary of the Analysis



So which move should we make? ;-)

Implementation?



MinMax algorithm

- Traverse “game tree”:
  - *Enumerate all possible moves at each node.*
  - *The children of each node are the positions that result from making each move. A leaf is a position that is won or drawn for some side.*

- Assume that we pick the best move for us, and the opponent picks the best move for him (causes most damage to us)

- Pick the move that maximizes the minimum amount of success for our side.

MinMax Algorithm

- Tic-Tac-Toe: three forms of success: Win, Tie, Lose.
  - *If you have a move that leads to a Win make it.*
  - *If you have no such move, then make the move that gives the tie.*
  - *If not even this exists, then it doesn't matter what you do.*

Extensions

- Challenges: In practice
  - *Trees too deep/large to explore (exponential complexity)*
  - *Opponent not always makes the 'best' choice*
  - *Randomness*
- Solution - Heuristics
  - *Rate nodes based on local information.*
  - *For example, in Chess "rate" a position by examining difference in number of pieces*

Heuristics in MinMax

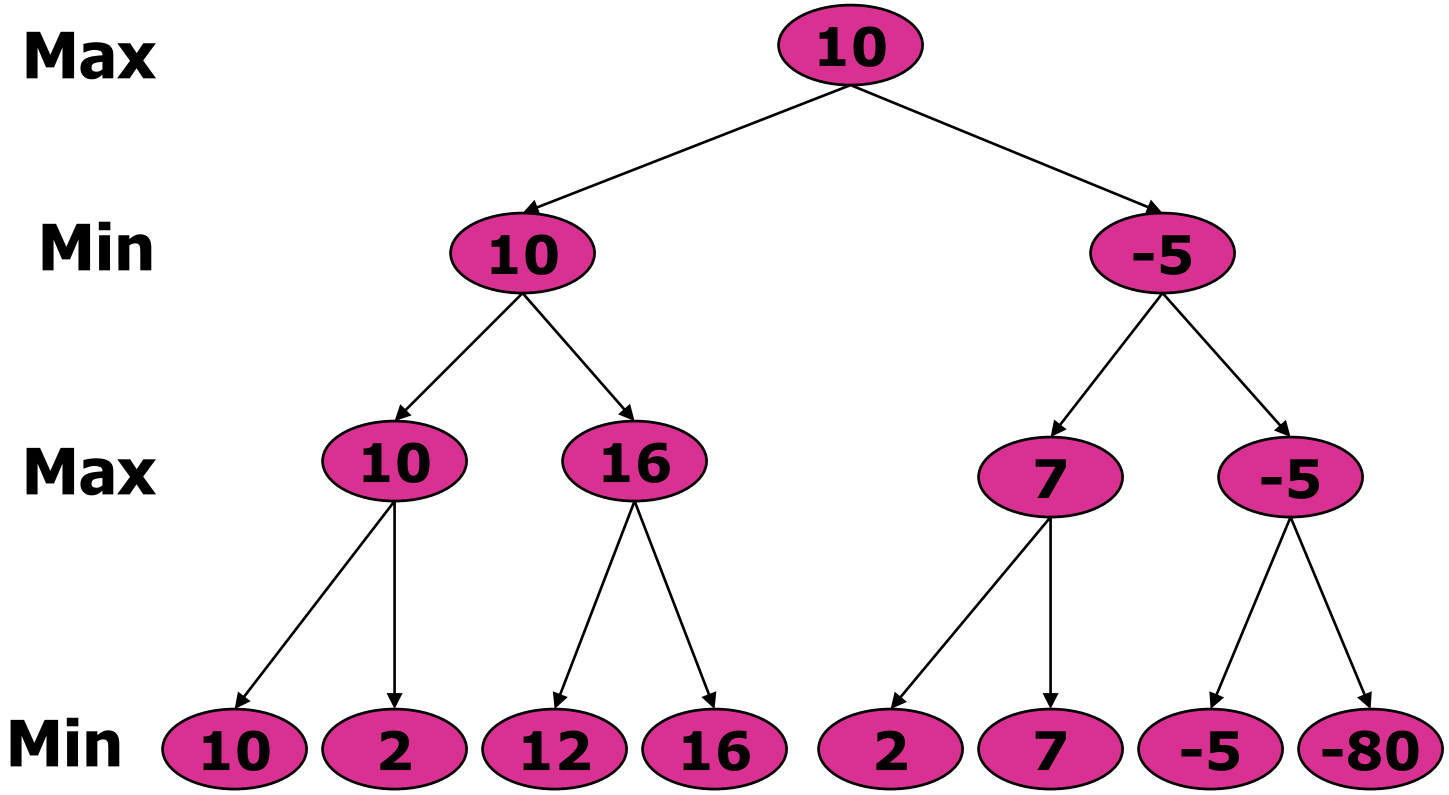
- Strategy that will let us cut off the game tree at fixed depth (layer)
- Apply heuristic scoring to bottom layer
  - *instead of just Win, Loss, Tie, we have a score.*
- For “our” level of the tree we want the move that yields the node (position) with highest score. For a “them” level “they” want the child with the lowest score.

Self stuy: Pseudocode

```
int Minimax(Board b, boolean myTurn, int depth) {
      if (depth==0)
            return b.Evaluate(); // Heuristic
      for(each possible move i)
            value[i] = Minimax(b.move(i), !myTurn,
depth-1);
      if (myTurn)
            return array_max(value);
      else
            return array_min(value);
}
```

Note: we don't use an explicit tree structure.
However, the pattern of recursive calls forms a tree on the call stack.

Real Minimax Example



Evaluation function applied to the leaves!

Alpha Beta Pruning

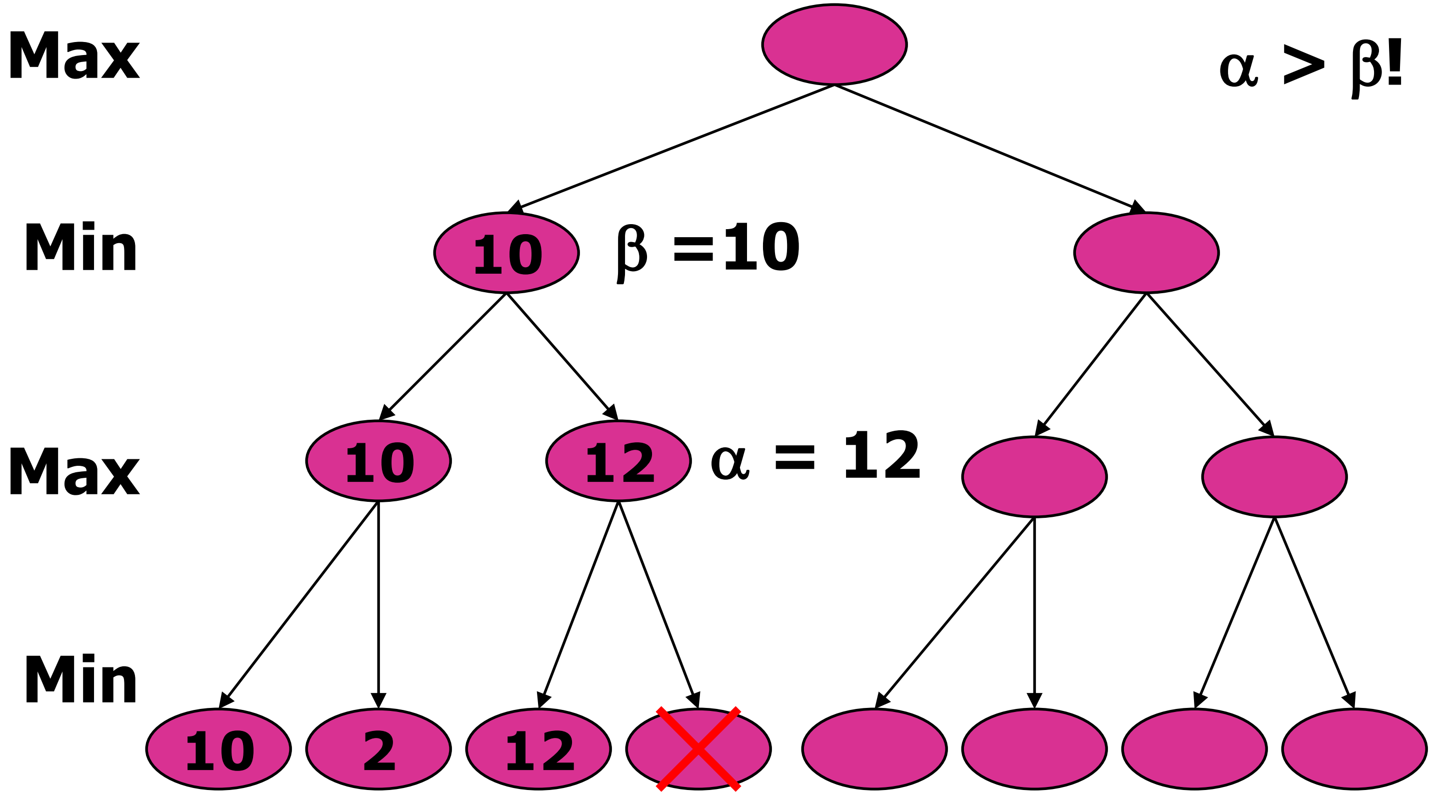
Idea: Track "window" of expectations.

Use two variables

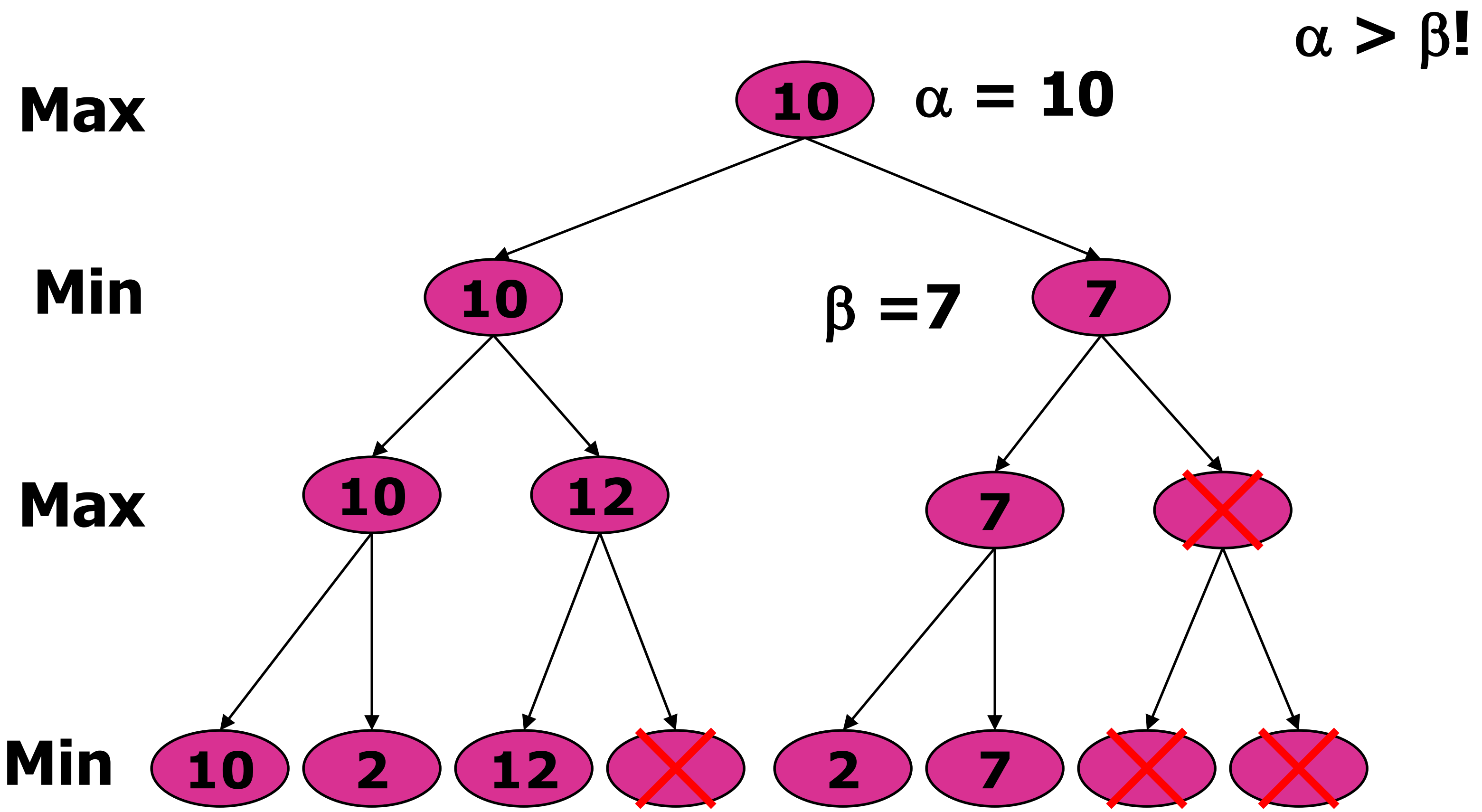
- α – Best score so far at a **max** node: increases
  - *At a child **min** node:*
    - Parent wants **max**. To affect the parent's current α**,** our β cannot drop below α**.**
  - *If β ever gets less:*
    - Stop searching further subtrees *of that child*. They do not matter!
- β – Best score so far at a **min** node: decreases
  - *At a child **max** node.*
    - Parent wants **min**. To affect the parent's current β**,** our α cannot get above the parent's β**.**
  - *If α gets bigger than β:*
    - Stop searching further subtrees *of that child*. They do not matter!

Start with an infinite window ($\alpha = -\infty$, $\beta = \infty$)

Alpha Beta Example I



Alpha Beta Example II



Self stuy: Pseudo Code

```
int AlphaBeta(Board b, boolean myTurn, int depth, int alpha, int beta) {
   if (depth==0)
      return b.Evaluate(); // Heuristic
   if (myTurn) {
      for(each possible move i && alpha < beta)
         alpha  = max(alpha,AlphaBeta(b.move(i),!myTurn,depth-1,alpha,beta));
      return alpha;
   }
   else {
      for(each possible move i && alpha < beta)
         beta  = min(beta,AlphaBeta(b.move(i), !myTurn, depth-1,alpha,beta));
      return beta;
   }
}
```

Debugging

Debugging

- ***There will be bugs…***

- ***Strategies for Fixing?***

Debugging

- ***There will be bugs…***

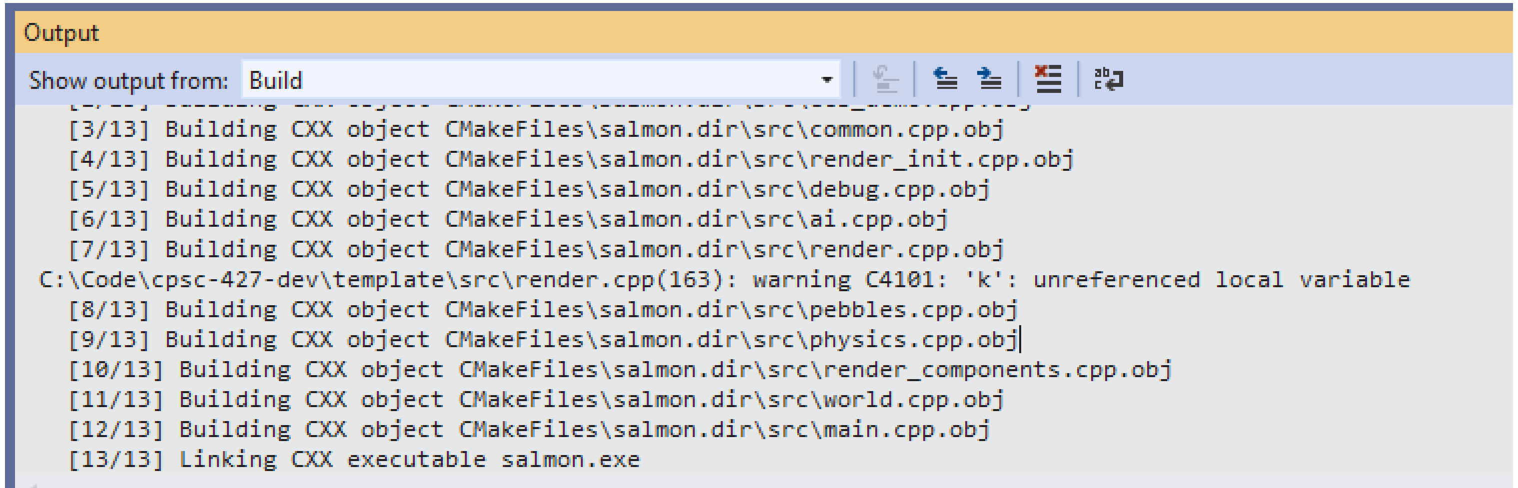
- ***Strategies for Fixing?***
- Anticipate
- Reproduce
  - *Things get terribly difficult if randomness is involved!*
- Localize
- Use proper debugging tools

Debugging:Strategies for Fixing?

- Anticipate I
  - *Unit tests*
  - *Logging*
  - *Explicit tests for "what can go wrong" (assert)*
    - Anything that can go wrong will go wrong… at the worst possible time
  - *State/play saving and loading speeds up debugging*
  - *Visual testing (early)*
  - *Avoid randomness (use seed for rnd)*
- Reproduce
- Localize
- Use proper debugging tools

Debugging: Strategies for Fixing?

- Anticipate II: *your compiler (with –Wall enabled) is your friend*
  - *“This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid”*
- Reproduce
- Localize
- Use proper debugging tools



Debugging

- ***Strategies for Fixing?***
- Anticipate
- Reproduce
  - *When does it happen?*
  - *Logging + unit tests*
  - *Record/load gameplay*
- Localize
- Use proper debugging tools

Debugging

- ***Strategies for Fixing?***
- Anticipate
- Reproduce
- Localize
  - *In time: version control*
  - *In place: logging*
    - Divide and Conquer
  - *Minimal trigger input*
  - *Don't guess; measure*
- Use proper debugging tools

Debugging

- ***Strategies for Fixing?***
- Anticipate
- Reproduce
- Localize
- Use proper debugging tools
  - *Run with debug settings on*
  - *Run within a debugger*
    - Set breakpoints
    - Examine internal state
  - *Learn debugger options*

Demo

Debugging (From Waterloo ECE 155)

More (Human Factor) Strategies

- Take a Break/Sleep on it
- Code Review
    - Look through code
    - Walk someone through the code

Debugging

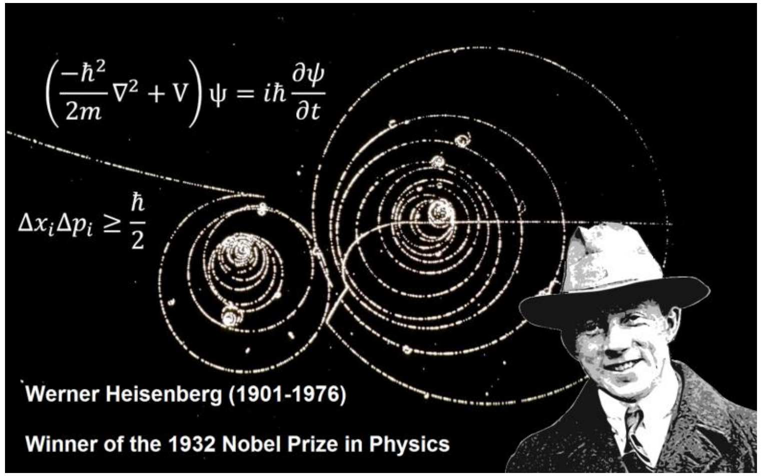
More (Human Factor) Strategies

- Question assumptions
- Minimize randomness
  - Use same seed
- Check boundary conditions
- Disrupt parallel computations

Debugging (From Waterloo ECE 155)

More Strategies

- Know your enemy: Types of bugs
  - Standard bug (reproducible)
  - Sporadic (need to chase – right input combo)
  - Heisenbug
    - Memory (not initialized or stepped on)
    - Parallel execution
    - Optimization



Hard Bugs (cheat sheet)

- *Bug occurs in Release but not Debug*
  - Uninitialized data or optimization issue
- *Bug disappears when changing something innocuous*
  - Timing or memory overwrite problem
- *Intermittent problems*
  - Record as much info when it does happen
- *Unexplainable behavior*
  - Retry, Rebuild, Reboot, Reinstall
- *Internal compiler errors (not likely)*
  - Full rebuild, divide and conquer, try other machines
- *Suspect it's not your code (not likely)*
  - Check for patches, updates, or reported bugs

Physics

Physics-Based Simulation

- Movement governed by forces
- Simple
  - *Independent particles*
- Complex
  - *Correct collisions, stacking, sliding 3D rigid bodies*
- Many many simulators!
  - *PhysX (Unity, Unreal), Bullet, Open Dynamics Engine, MuJoCo, Havok, Box2D, Chipmunk, OpenSim, RBDL, Simulink (MATLAB), ADAMS, SD/FAST, DART etc…*

Examples

- Particle systems
  - *Fire, water, smoke, pebbles*
- Rigid-body simulation
  - *Blocks, robots, humans*
- Continuum systems
  - *Deformable solids*
  - *Fluids, cloth, hair*
- Group movement
  - *Flocks, crowds*

Simulation Basics

Simulation loop…

1. ***Equations of Motion***
   - sum forces & torques
   - solve for accelerations: $\vec{\boldsymbol{F}} = \boldsymbol{ma}$
2. ***Numerical integration***
   - update positions, velocities
3. ***Collision detection***
4. ***Collision resolution***

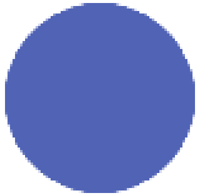
Basic Particle Simulation (first try)

Forces only $\vec{F} = ma$

$$d_t = t_{i+1} - t_i$$

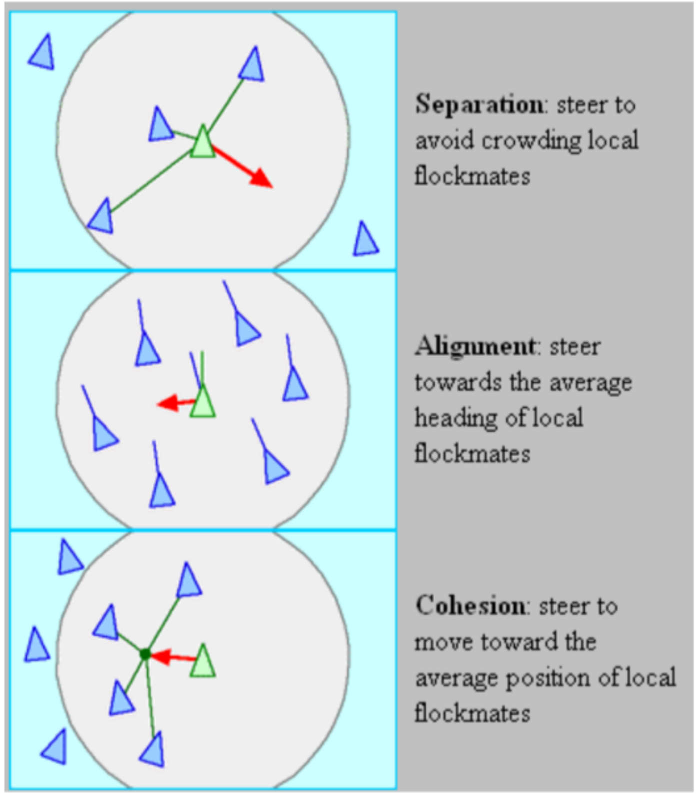
$$\vec{v}_{i+1} = \vec{v}(t_i) + (\vec{F}(t_i)/m)d_t$$

$$\vec{p}_{i+1} = \vec{p}(t_i) + \vec{v}(t_{i+1})d_t$$

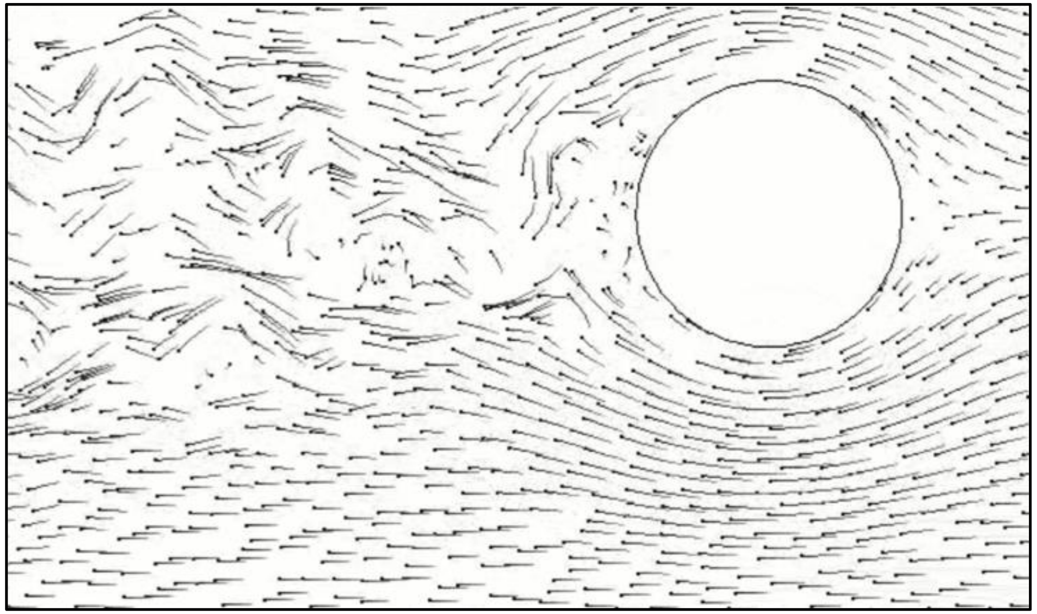


Proxy Forces

- **Behavior forces:**
  flocking birds, schooling fish, etc.
  [“Boids”, Craig Reynolds, SIGGRAPH 1987]

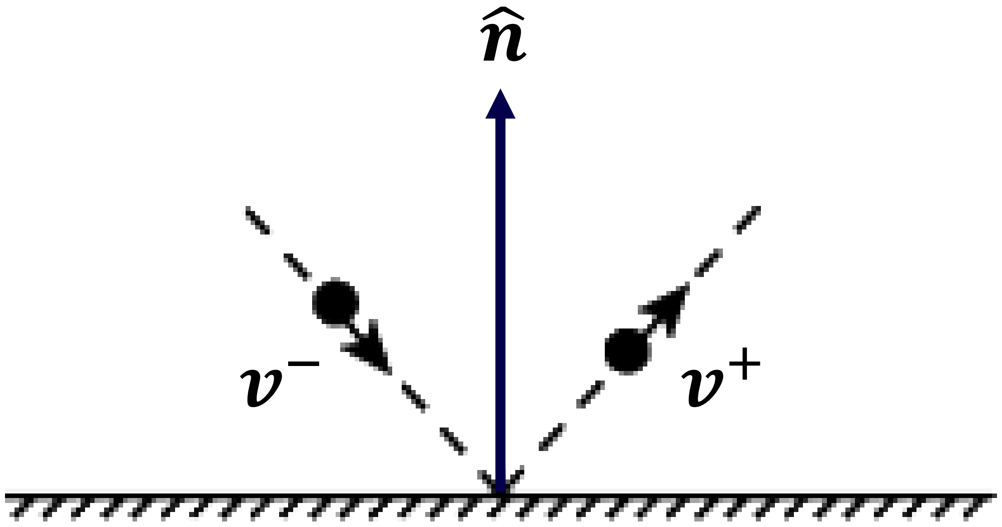


- **Fluids**
  [“Curl Noise for Procedural Fluid Flow” R. Bridson, J. Hourihan, M. Nordenstam, Proc. SIGGRAPH 2007]



Particle-Plane Collisions

- ***Apply an impulse of magnitude j***
  - Inversely proportional to mass of particle
- ***In direction of normal***



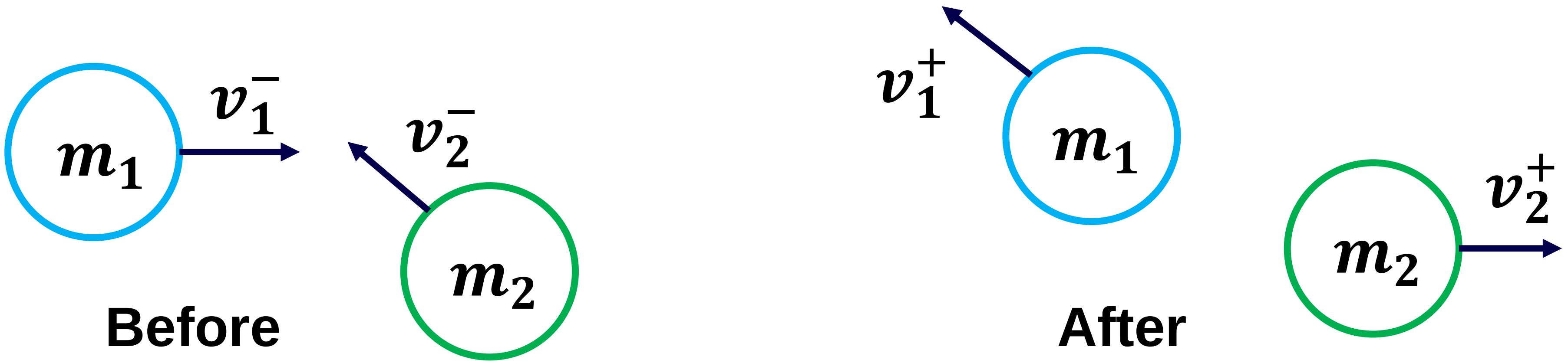
$$j = (1+\epsilon)m$$

$$\vec{j} = j\,\hat{n}$$

$$v^+ = \frac{\vec{j}}{m} + v^-$$

Particle-Particle Collisions (radius=0)

- Particle-particle **frictionless elastic impulse response**



- Momentum is **preserved**

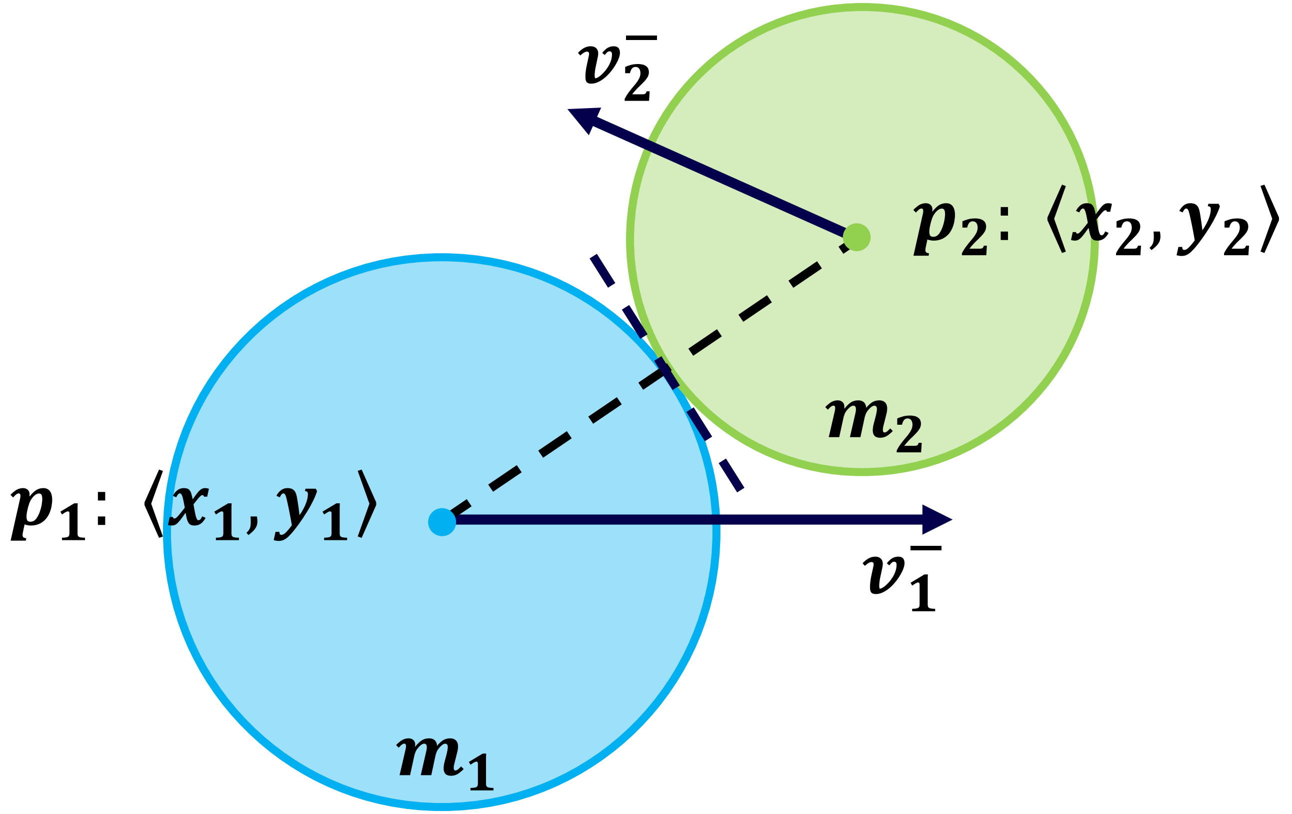
$$m_1 v_1^- + m_2 v_2^- = m_1 v_1^+ + m_2 v_2^+$$

- Kinetic energy is **preserved**

$$\tfrac{1}{2} m_1 {v_1^-}^2 + \tfrac{1}{2} m_2 {v_2^-}^2 = \tfrac{1}{2} m_1 {v_1^+}^2 + \tfrac{1}{2} m_2 {v_2^+}^2$$

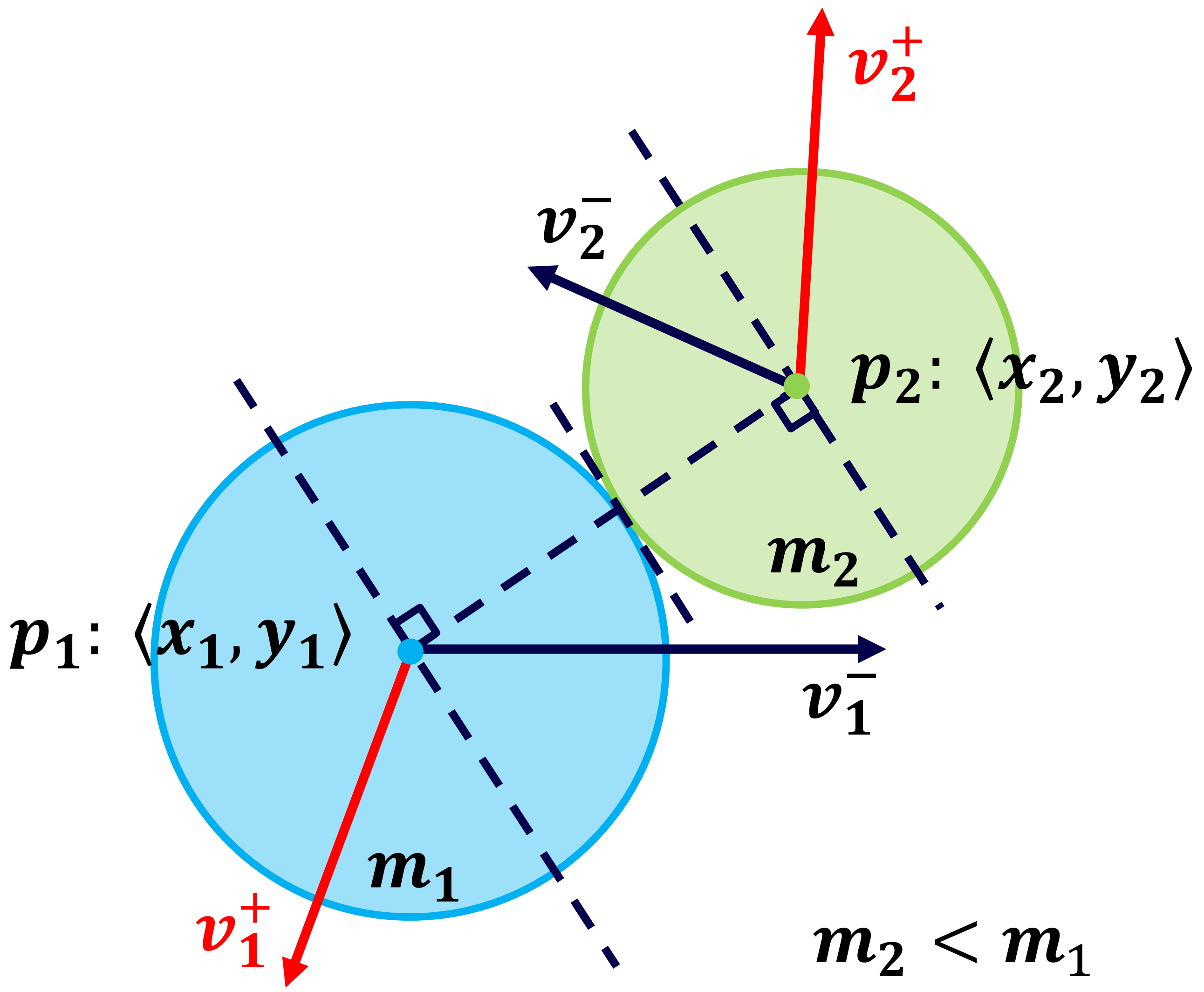
Particle-Particle Collisions (radius >0)

- **What we know…**
  - ***Particle centers***
  - ***Initial velocities***
  - ***Particle Masses***
- **What we can calculate…**
  - ***Contact normal***
  - ***Contact tangent***



Particle-Particle Collisions (radius >0)

- **Impulse direction reflected across tangent**
- **Impulse magnitude proportional to mass of other particle**



Particle-Particle Collisions (radius >0)

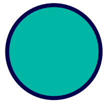
- **More formally…**

$$\boldsymbol{v}_1^+ = \boldsymbol{v}_1^- - \frac{2\boldsymbol{m}_2}{\boldsymbol{m}_1 + \boldsymbol{m}_2} \frac{\langle \boldsymbol{v}_1^- - \boldsymbol{v}_2^- \rangle \cdot \langle \boldsymbol{p}_1 - \boldsymbol{p}_2 \rangle}{\|\boldsymbol{p}_1 - \boldsymbol{p}_2\|^2} \langle \boldsymbol{p}_1 - \boldsymbol{p}_2 \rangle$$

$$\boldsymbol{v}_2^+ = \boldsymbol{v}_2^- - \frac{2\boldsymbol{m}_1}{\boldsymbol{m}_1 + \boldsymbol{m}_2} \frac{\langle \boldsymbol{v}_2^- - \boldsymbol{v}_1^- \rangle \cdot \langle \boldsymbol{p}_2 - \boldsymbol{p}_1 \rangle}{\|\boldsymbol{p}_2 - \boldsymbol{p}_1\|^2} \langle \boldsymbol{p}_2 - \boldsymbol{p}_1 \rangle$$

Rigid Body Dynamics

- **From particles to rigid bodies…**



Particle

Rigid body

$$state = \begin{cases} \vec{x}\ position \\ \vec{v}\ velocity \end{cases}$$

$\mathbb{R}^4$ in 2D
$\mathbb{R}^6$ in 3D

$$state = \begin{cases} \vec{x}\ position \\ \vec{v}\ velocity \\ q, R\ rotation\ matrix\ 3x3 \\ \vec{w}\ angular\ velocity \end{cases}$$

$\mathbb{R}^{12}$ in 3D