Two-player games

www.npr.org

Setup

@Helge: Pressed record?

@Class: Logged into iClicker cloud?

Overview

First half:

- ***Shortest paths cont.***
- ***Two-player games***

… all about traversing trees efficiently

+ Some debugging tips

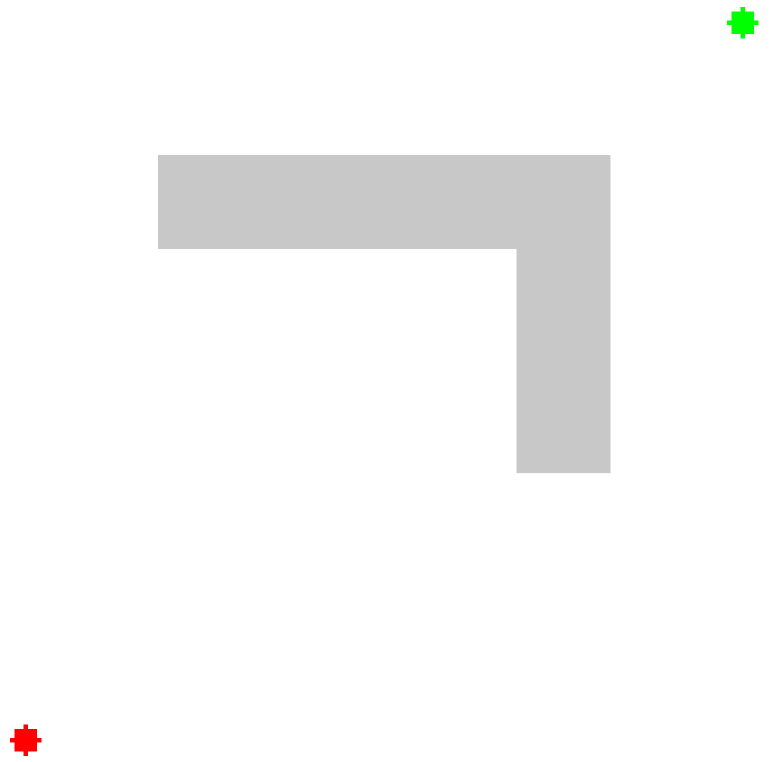
Second half:

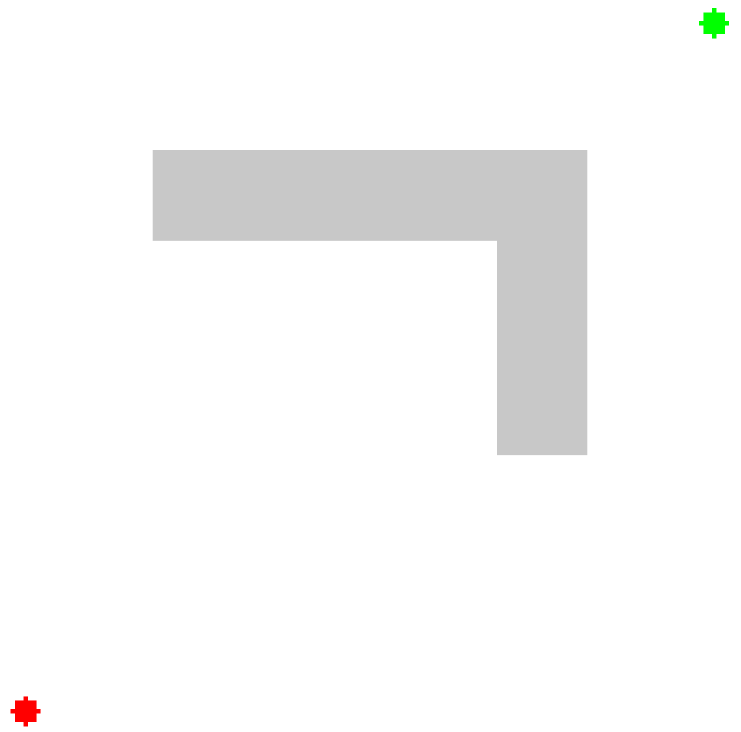
- ***Physical simulation basics***
  - *setting and definitions*
- ***Efficient & precise simulation***
  - *today: what can go wrong?*

… the core of every game?

End of the day: ***be able to implement efficient shortest path, two-player AI, and to simulate flying pebbles (for A3!)***

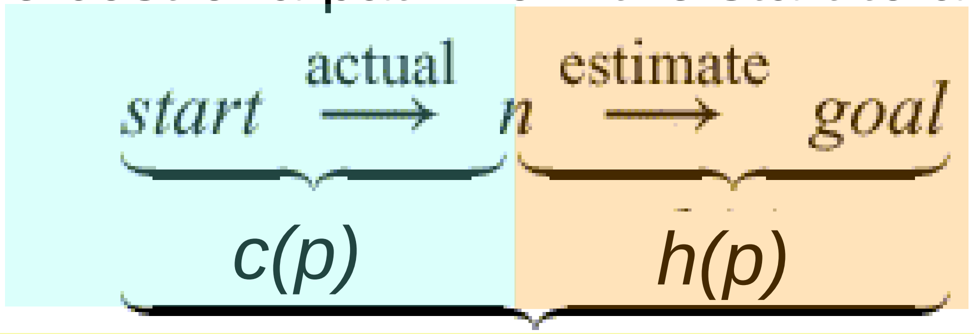
Breadth-first vs. A*





A* Search

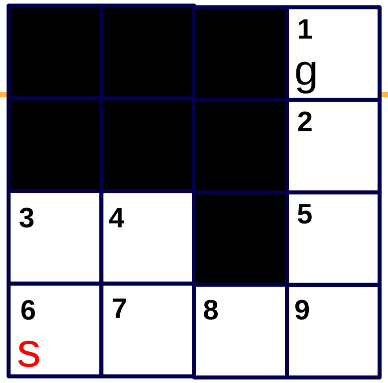
- A* search takes into account both
  - *c(p)* = cost of path *p* to current node
  - *h(p)* = heuristic value at node *p (estimated "remaining" path cost)*
- Let *f(p) = c(p) + h(p).*
  - *f(p)* is an estimate of the cost of a path from the start to a goal via *p*.



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* Example

Frontier (open list)



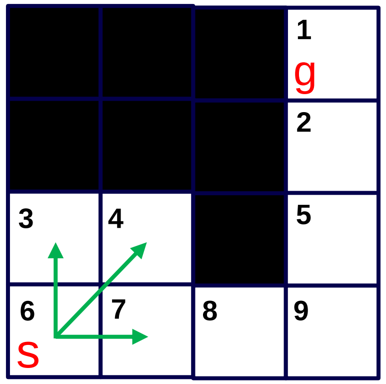
Init:

- ***Put starting node on open list:*** Lo = {6}
- ***Set its cost to 0:*** c[6] = 0
- ***Set closed list to empty list:*** Lc = {}

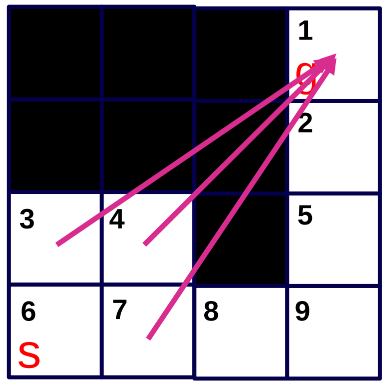
Step 1:

- ***Find node with smallest f on the list, call it q:*** q = 6
- ***Find q's "successors":*** sucs = {3,4,7}
- ***For each successor u:*** for u in sucs …
  - c(u) = c(q) + d(q,u)   c[3] = c[6] + 1 = 1
    c[4] = c[6] + 1.4 = 1.4
    c[7] = c[6] + 1 = 1
  - h(u) = d(g, u)   h[3] = 3.6   f[3] = c[3] + h[3] = 4.6
    f(u) = c(u) + h(u)   h[4] = 2.8   f[4] = c[4] + h[4] = 4.2
    h[7] = 3.6   f[7] = c[7] + h[7] = 4.6
- ***add successors to open list and move q to closed:***
  Lo = {3,4,7}; Lc = {6}

Step cost c



Heuristic dist. h

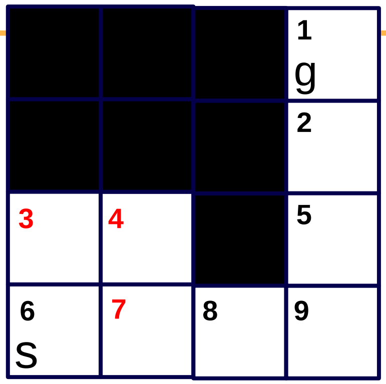


A* Example

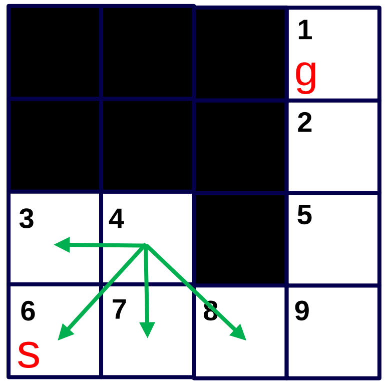
Step 2: Lo = {3,4,7}; Lc = {6}

- ***Find node with smallest f on Lo, call it q:***
  - f[3] = 4.6
    f[4] = 4.2 -> q = 4
    f[7] = 4.6
- ***Find q's "successors":*** sucs = {3,6,7,8}
- for u in sucs…
  - c_tmp[3] = c[4] + 1 = 2.4 > c[3] = 1, skip
    c_tmp[6] = c[4] + 1.4 = 2.8 > c[6] = 0, skip
    c_tmp[7] = c[4] + 1 = 2.4 > c[7] = 1, skip
    c_tmp[8] = c[4] + 1.4 = 2.4 not in Lo or Lc, select c[8] = c_tmp[8]
  - Update heuristic and estimated cost f:
    h[8] = 3.2
    f[8] = c[8] + h[8] = 5.6
- ***add successors to open list and move q to closed list:***
  Lo = {3,7,8}; Lc = {6,4}

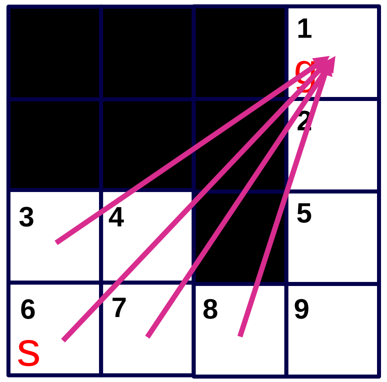
Frontier (open list)



Step cost c



Heuristic dist. h

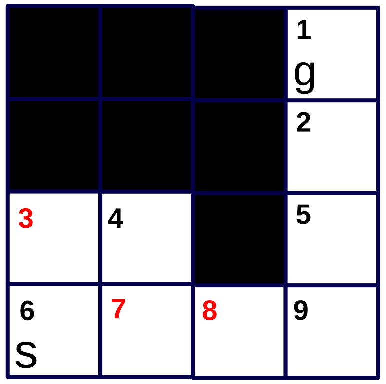


A* Example

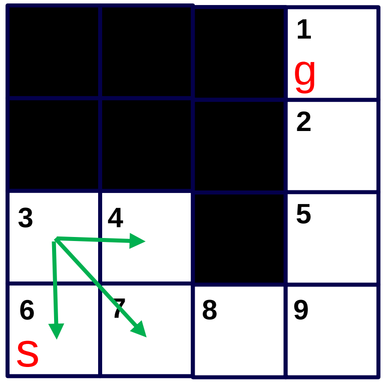
Step 3: Lo = {3,7,8}; Lc = {6,4}

- ***Find node with smallest f on Lo, call it q:***
  - f[3] = 4.6 -> q = 3
    f[7] = 4.6
    f[8] = 5.6
- ***Find q's "successors":*** sucs = {4,6,7}
- for u in sucs…
  - c_tmp[4] = c[3] + 1 = 2 > c[4] = 1.4, skip
    c_tmp[6] = c[3] + 1.4 = 2.4 > c[6] = 0, skip
    c_tmp[7] = c[3] + 1 = 2 > c[7] = 1, skip
- ***add successors to open list? no successors!***
- ***move q to closed list:***
  Lo = {7,8};
  Lc = {6,4,3}

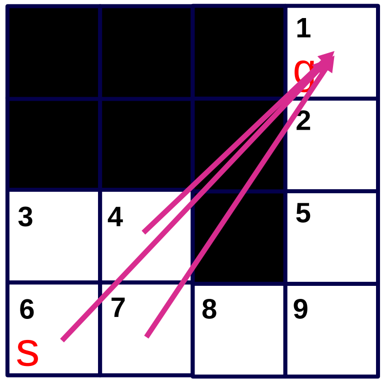
Frontier (open list)



Step cost c



Heuristic dist. h

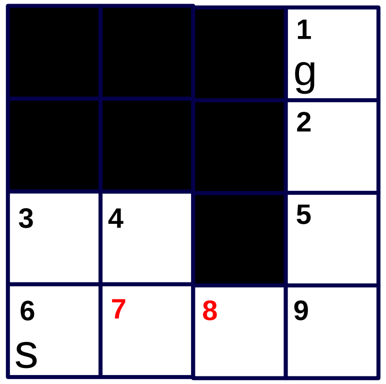


A* Example

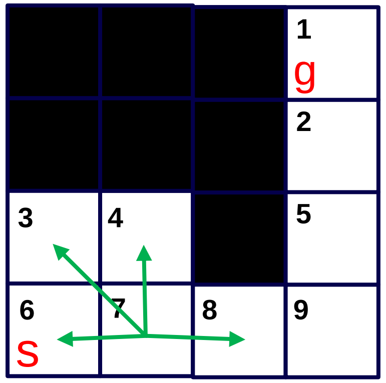
Step 4: Lo = {7,8}; Lc = {6,4,3}

- ***Find node with smallest f on Lo, call it q:***
  - f[7] = 4.6 -> q = 7
    f[8] = 5.6
- ***Find q's "successors":*** sucs = {3,4,6,8}
- **for u in sucs…**
  - c_tmp[3] = c[7] + 1.4 = 2.4 > c[3] = 1, skip
    c_tmp[4] = c[7] + 1 = 2 > c[4] = 1, skip
    c_tmp[6] = c[7] + 1 = 2 > c[6] = 0, skip
    c_tmp[8] = c[7] + 1 = 2 > c[8] = 2.4, select new c[8] = 2
- ***add successors to open list? Already there!***
- ***move q to closed list:***
  Lo = {8};
  Lc = {6,4,3,7}

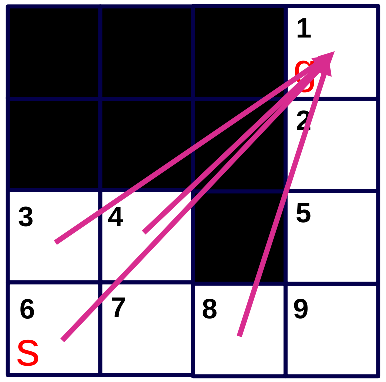
Frontier (open list)



Step cost c



Heuristic dist. h



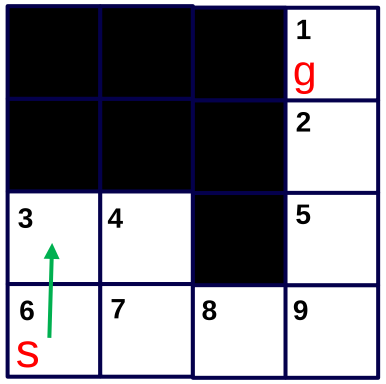
Keep track of your parents

- ***We neglected parent-child relation in previous slides…***

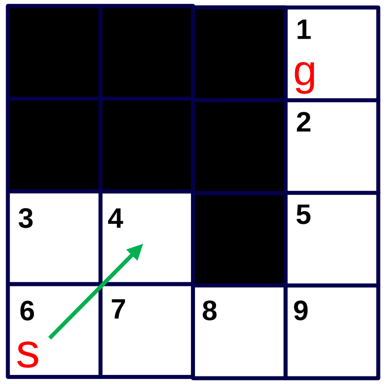
Lc = {6,4,3}

Lo = {8};

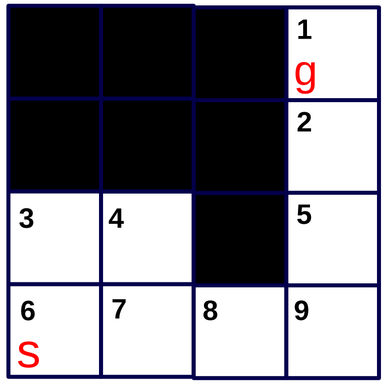
Path to 3



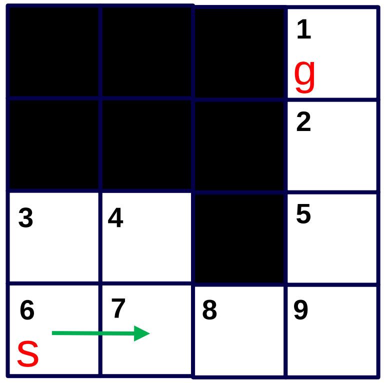
Path to 4



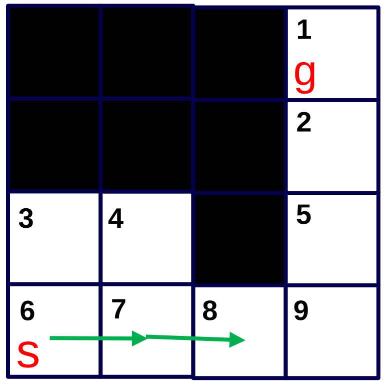
Path to 6



Path to 7



Path to 8



- ***Note, closed paths have no 'free' neighbors***
  - impassable or already visited from a shorter path

A* search

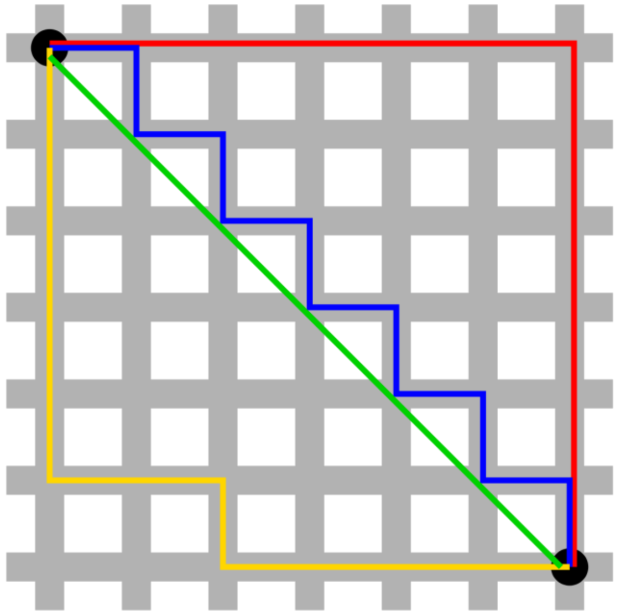
Key idea: H is a heuristic, and not the real distance:

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

Variants

- ***Randomness***

- ***Make the AI dump/non-perfect***
  - *How?*

- ***Different terrain types?***

Two-player games

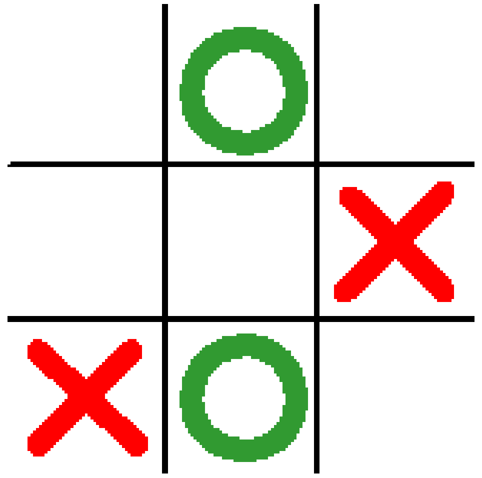
www.npr.org

Min-Max Trees

- Adversarial planning in a turn-taking environment
  - *Algorithm seeks to maximize our success* ***F***
  - *Adversary seeks to minimize* ***F***
  - $\boldsymbol{a_{we}} = \max_{\boldsymbol{we}} \min_{\boldsymbol{they}} \boldsymbol{F}(\boldsymbol{a_{we}}, \boldsymbol{a_{they}})$
- Key idea: at each step the algorithm selects the move that minimizes the highest (estimated) value of F the adversary can reach
  - *Assume the opponent does what is best*

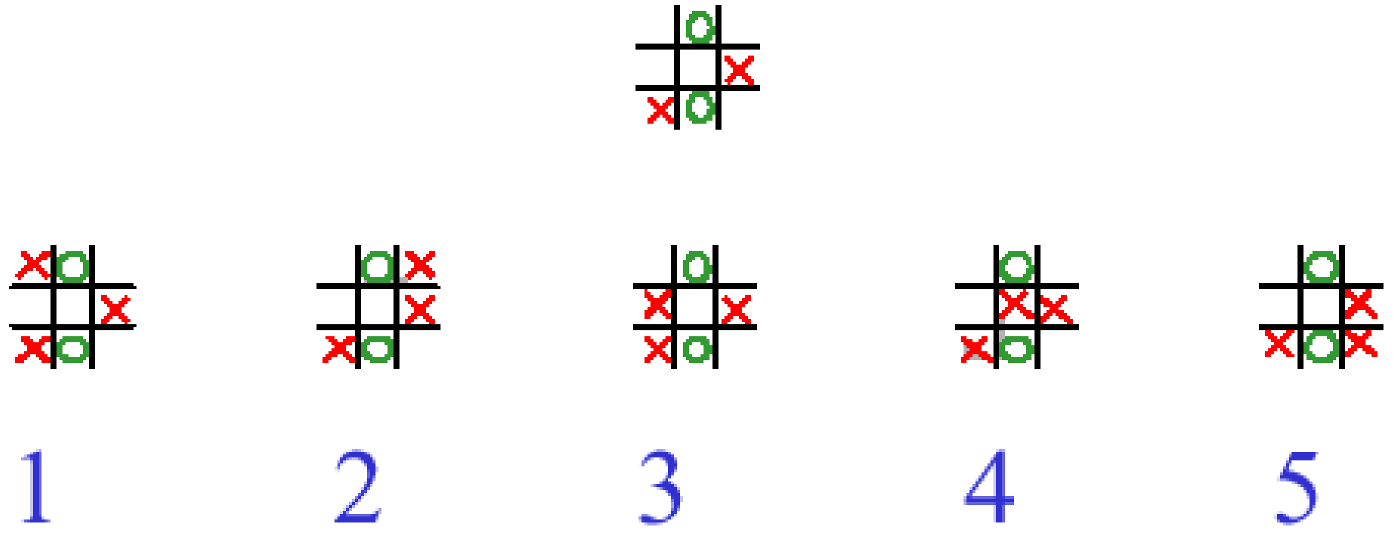
Example

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



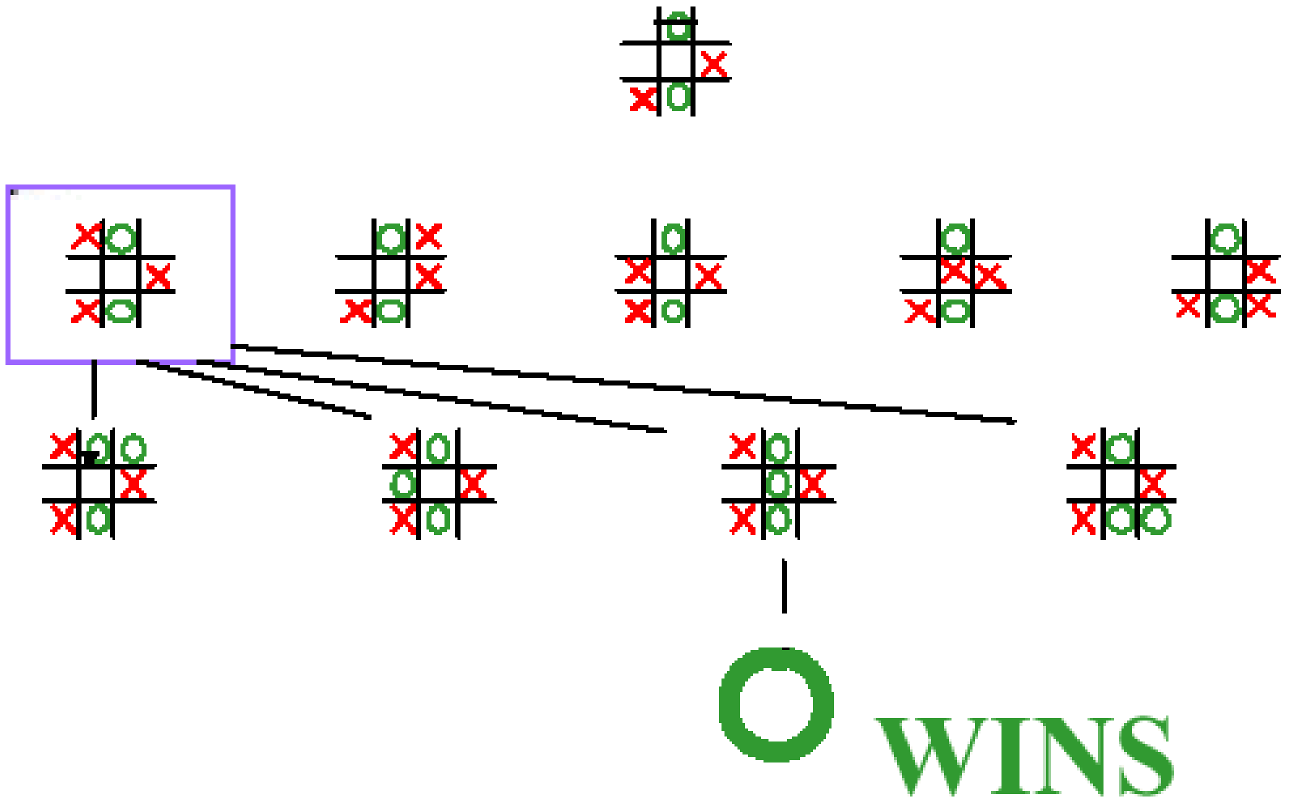
We are playing X, and it is now our turn.

Our options:



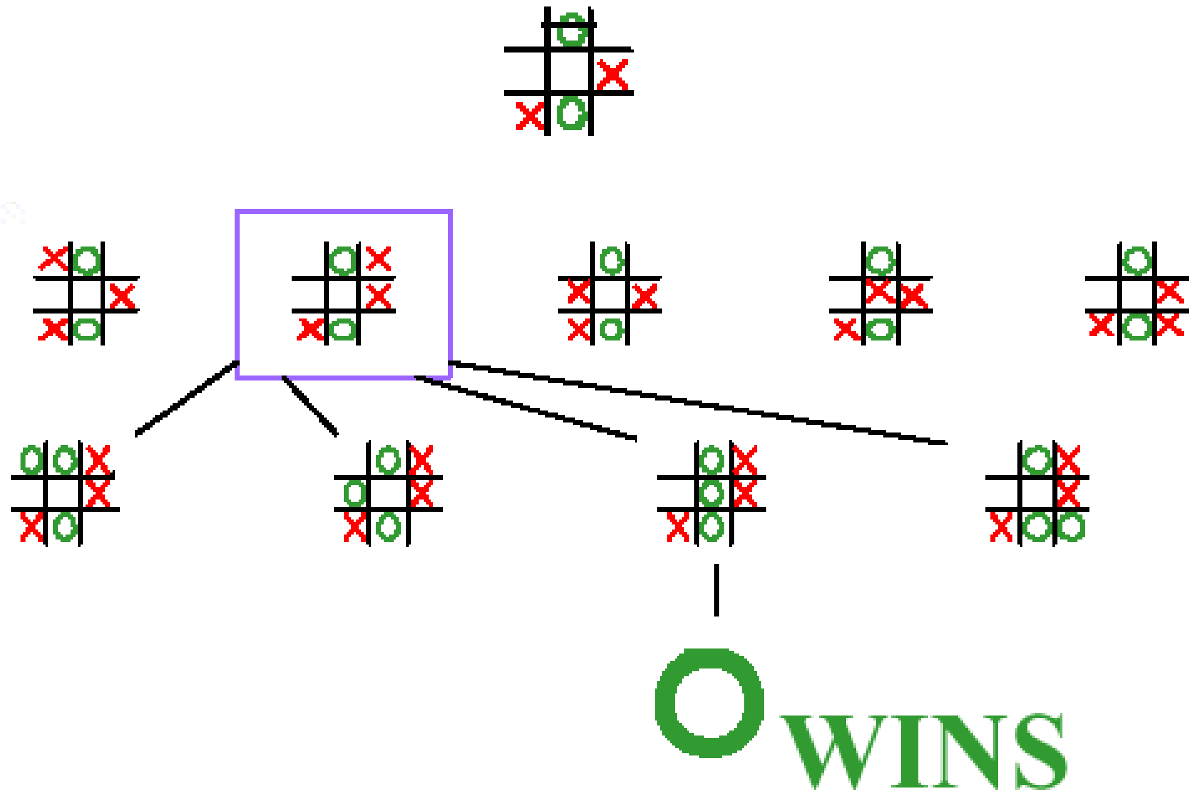
Number = position after each legal move

Opponent options



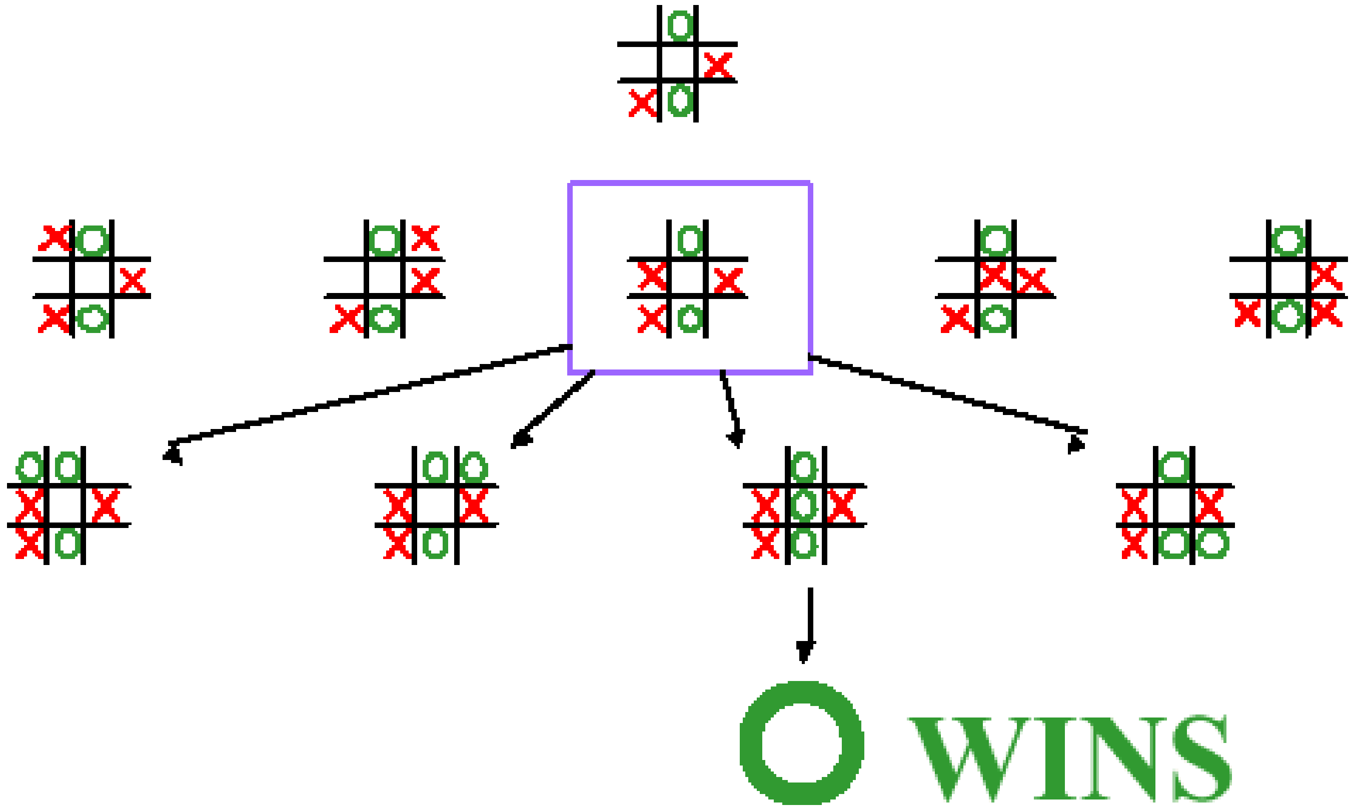
Here we are looking at all of the opponent responses to the first possible move we could make.

Opponent options

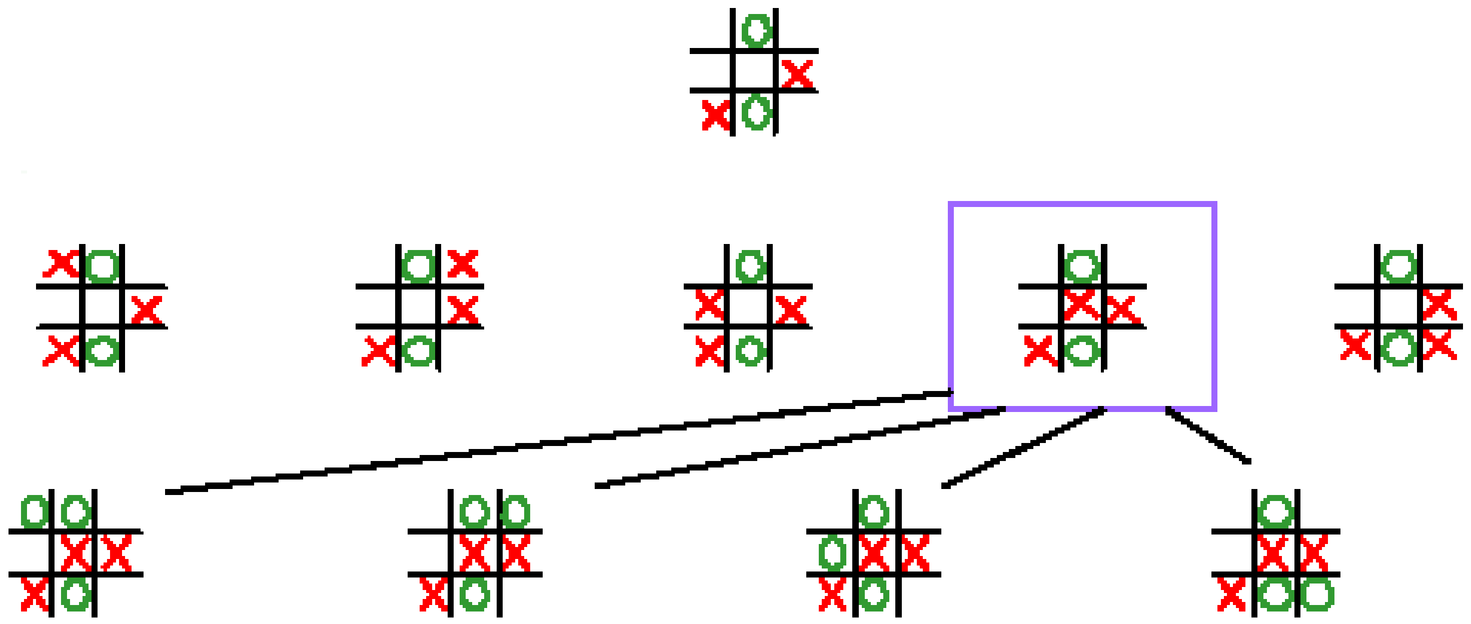


Opponent options after our second possibility. Not good again…

Opponent options

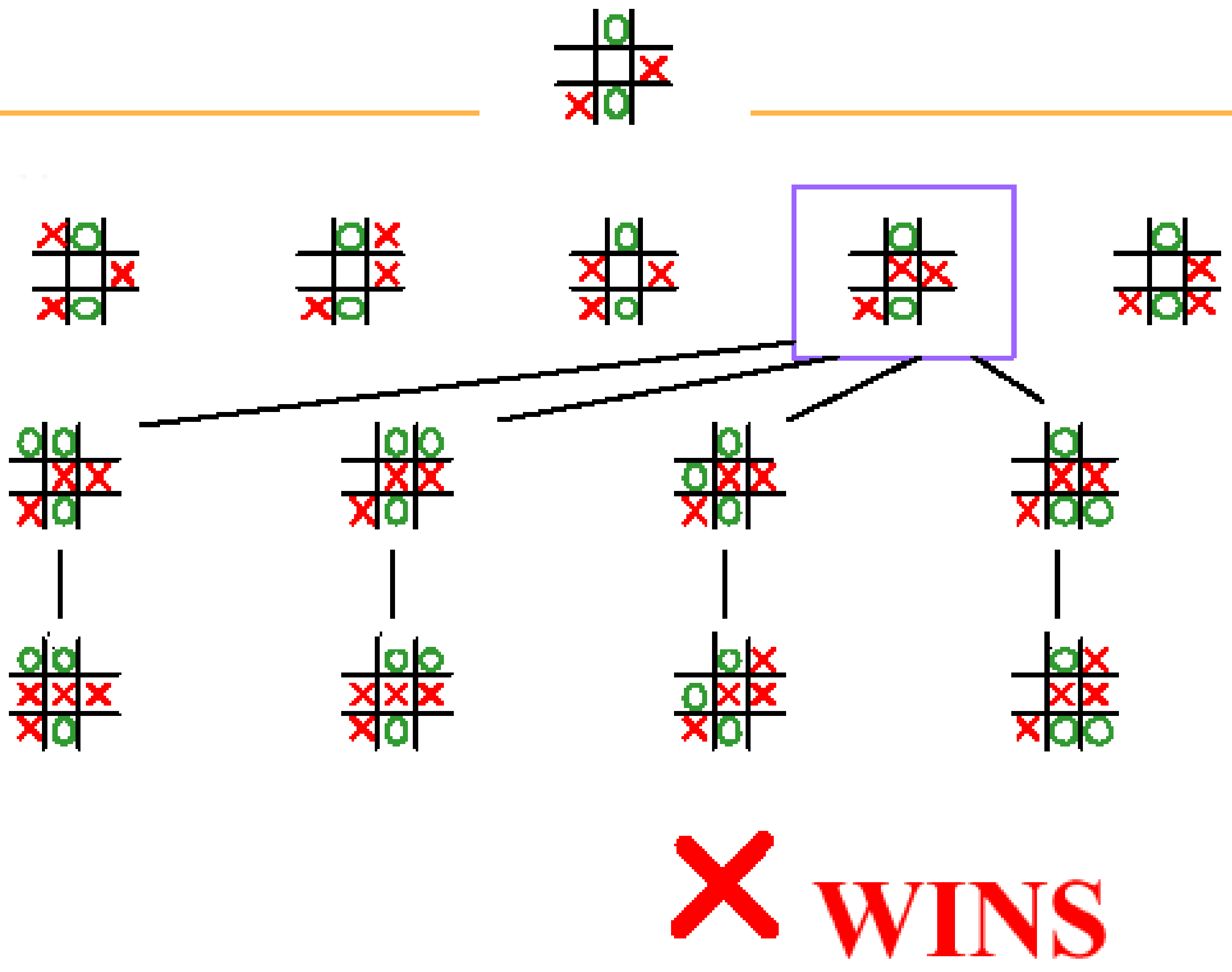


Opponent options => Our options



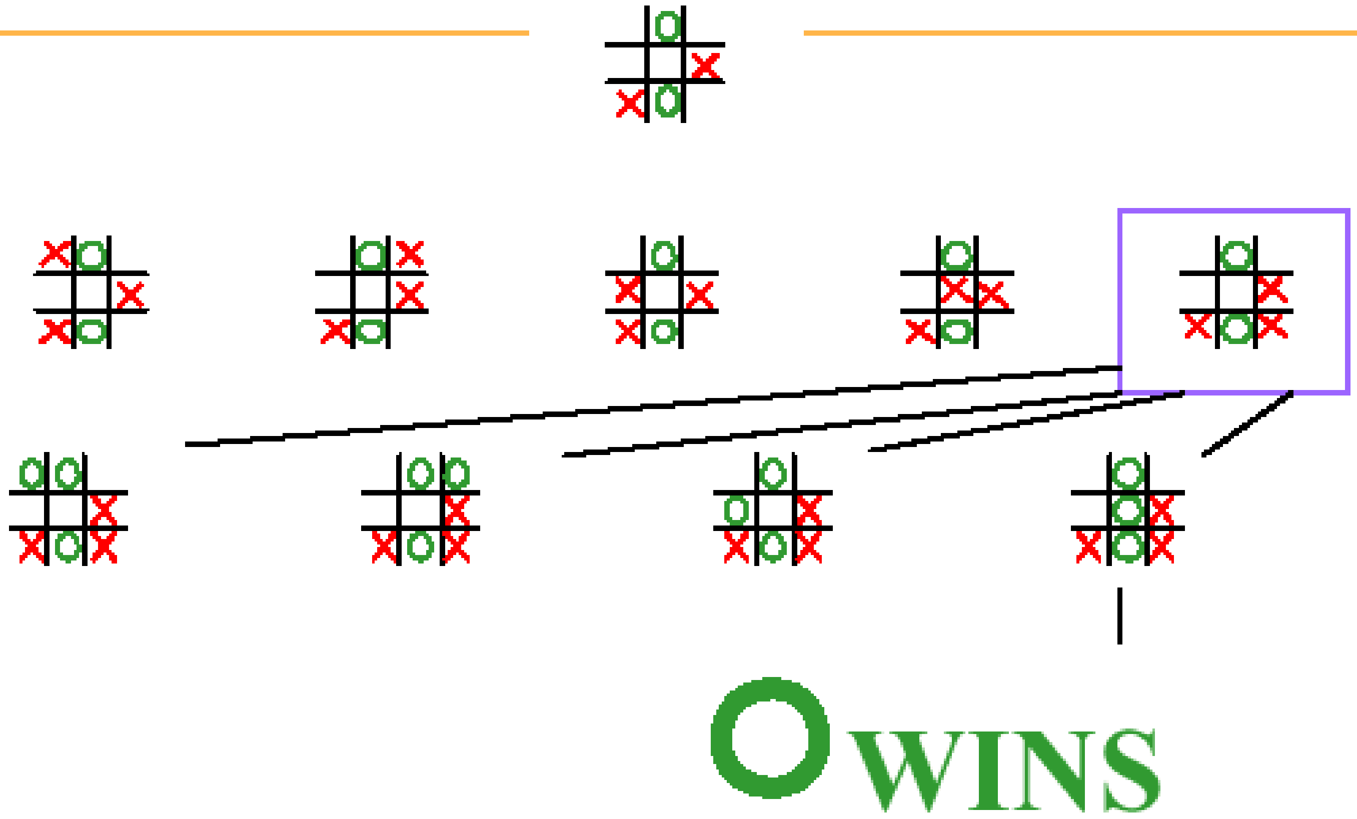
Now they don’t have a way to win on their next move. So now we have to consider our responses to their responses.

Our options



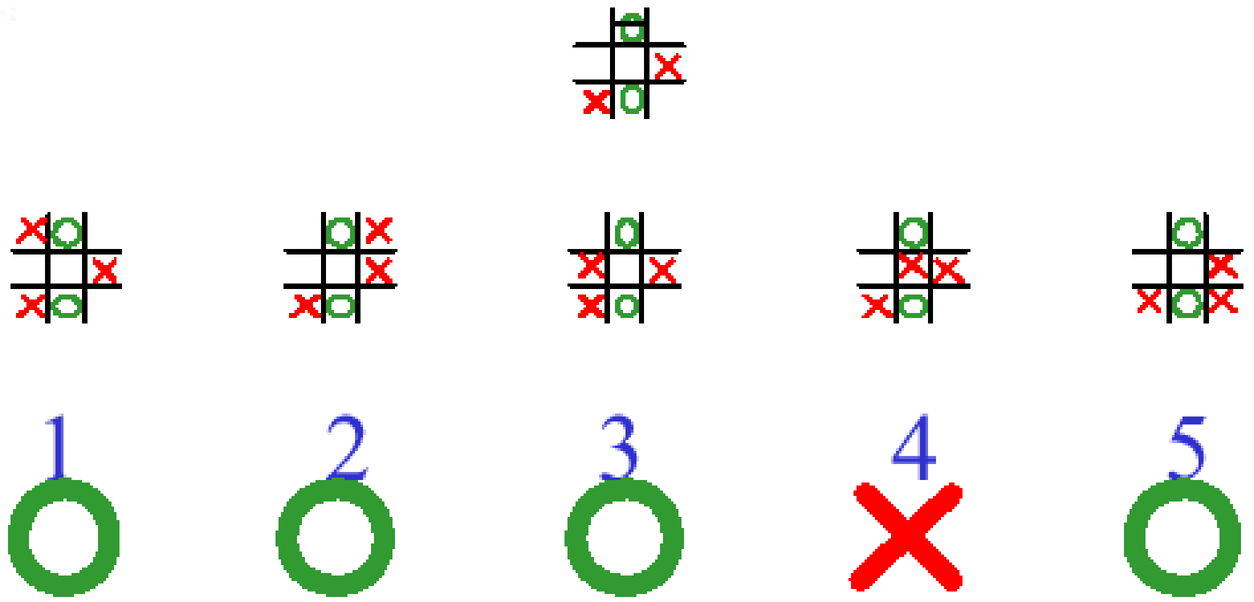
We have a win for any move they make.
Original position in purple is an X win.

Other options



They win again if we take our fifth move.

Summary of the Analysis



So which move should we make? ;-)

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 a draw or a win for some side.*

- Assume that we pick the best move for us, and the opponent picks the best move for them (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*
  - *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

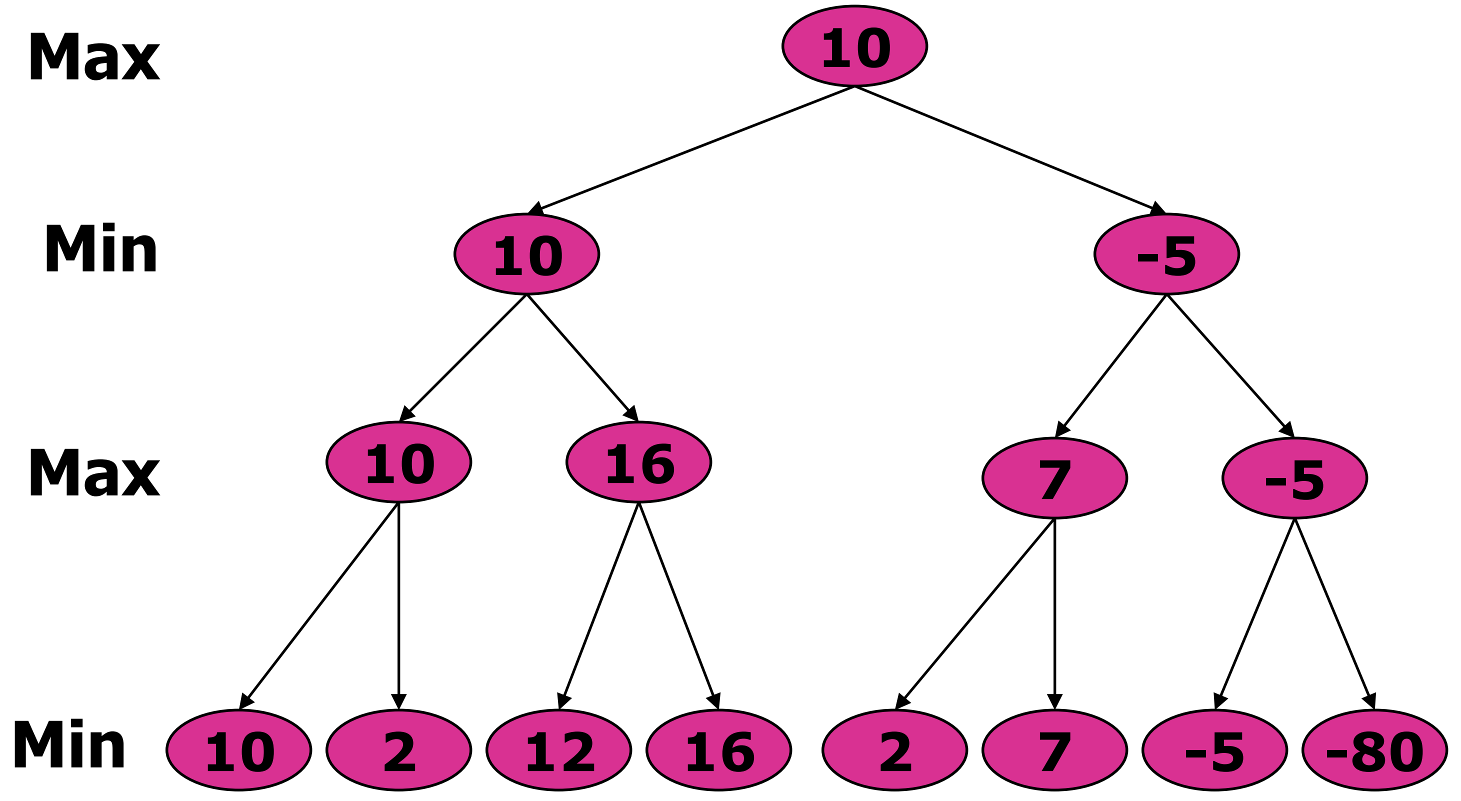
- Strategy that will let us cut off the game tree at fixed depth (layer)
- Apply heuristic scoring to bottom layer
  - *instead of just Win, Loss, Tie, we have a score.*
- For “our” level of the tree we want the move that yields the node (position) with highest score. For a “them” level “they” want the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

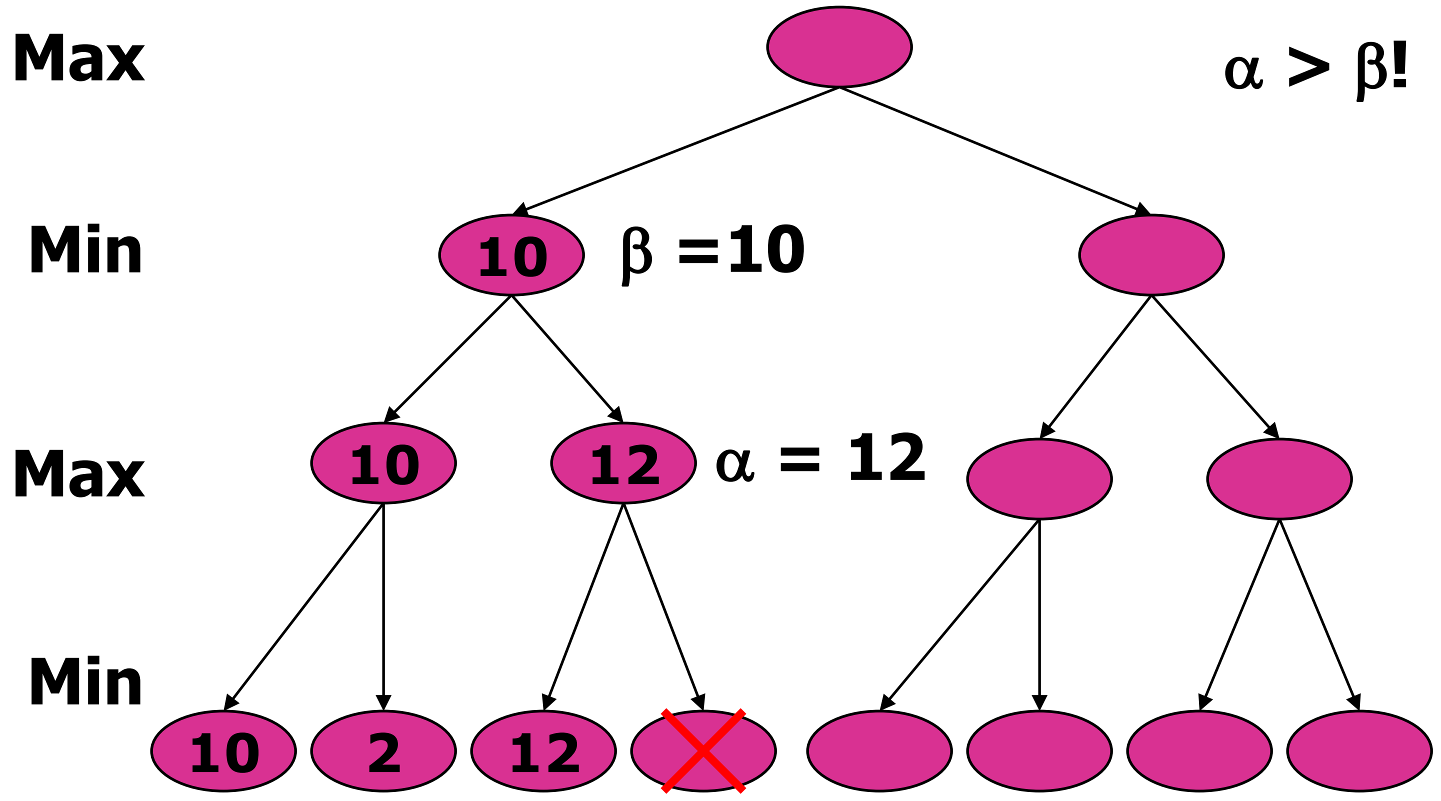
Note: we don't use an explicit tree structure.
However, the pattern of recursive calls forms a tree on the call stack.

Real Minimax Example



Evaluation function applied to the leaves!

Pruning Example



Self stuy: Alpha Beta Pruning

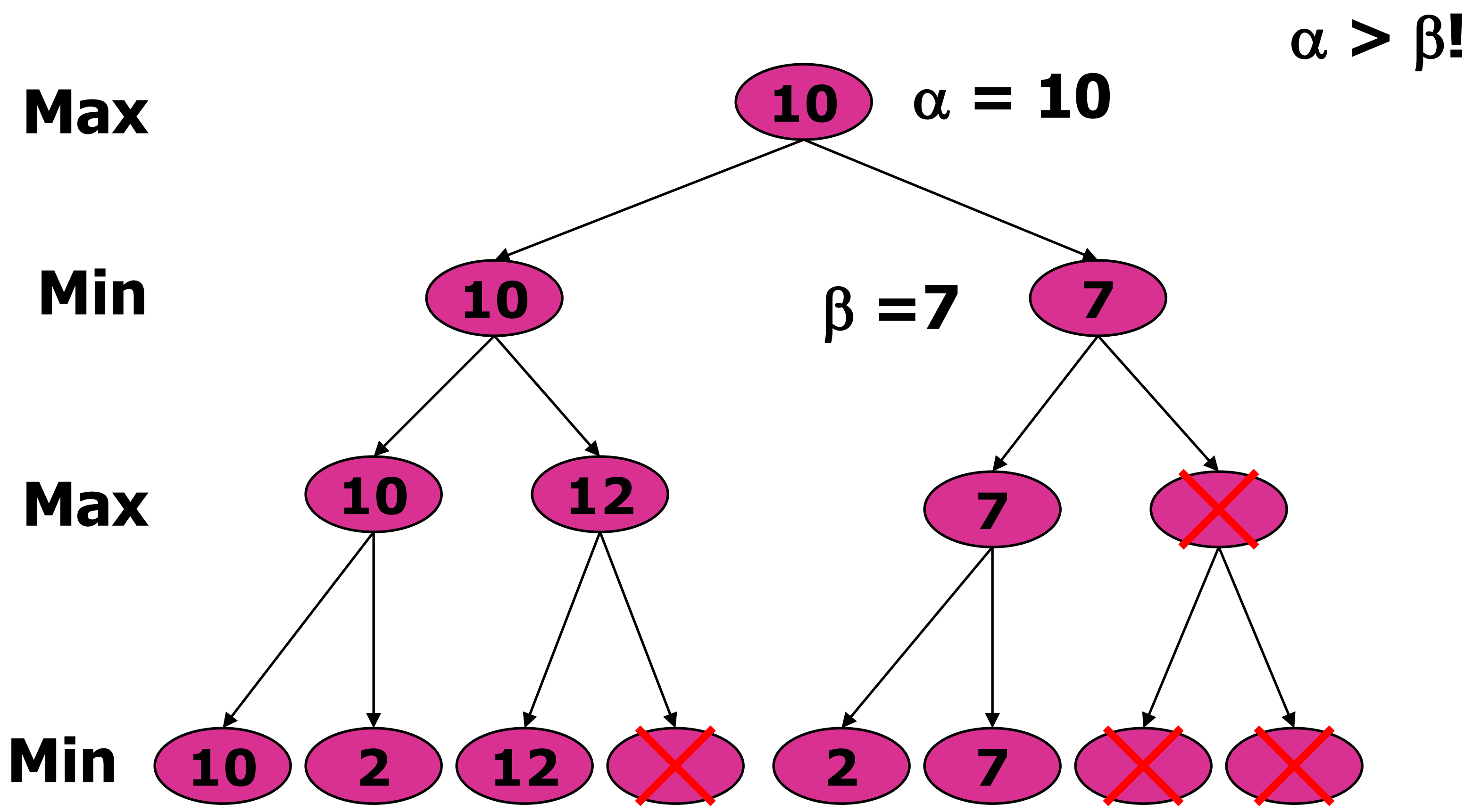
Idea: Track "window" of expectations.

Use two variables

- α – Best score so far at a **max** node ('our choice'): 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 ('their choice'): 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 ($\alpha = -\infty$, $\beta = \infty$)

Self stuy: 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;
   }
}
```

Variants

- ***More than two players?***

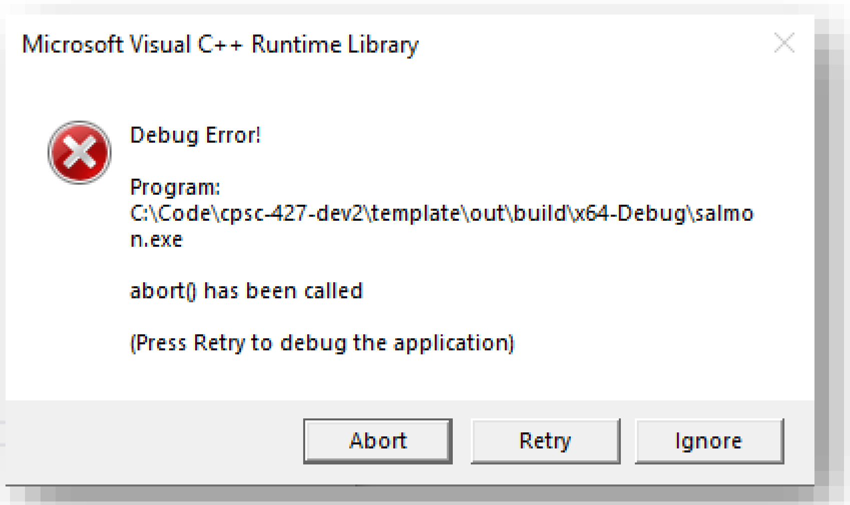
- ***More than two choices?***

- ***Opponent does not select best move?***

Debugging

Easy bugs

- ***Program crashes*** ☹ **?**
  - *This is likely an easy one ☺ !*
    - You know where the bug came from!
      Check your call stack!



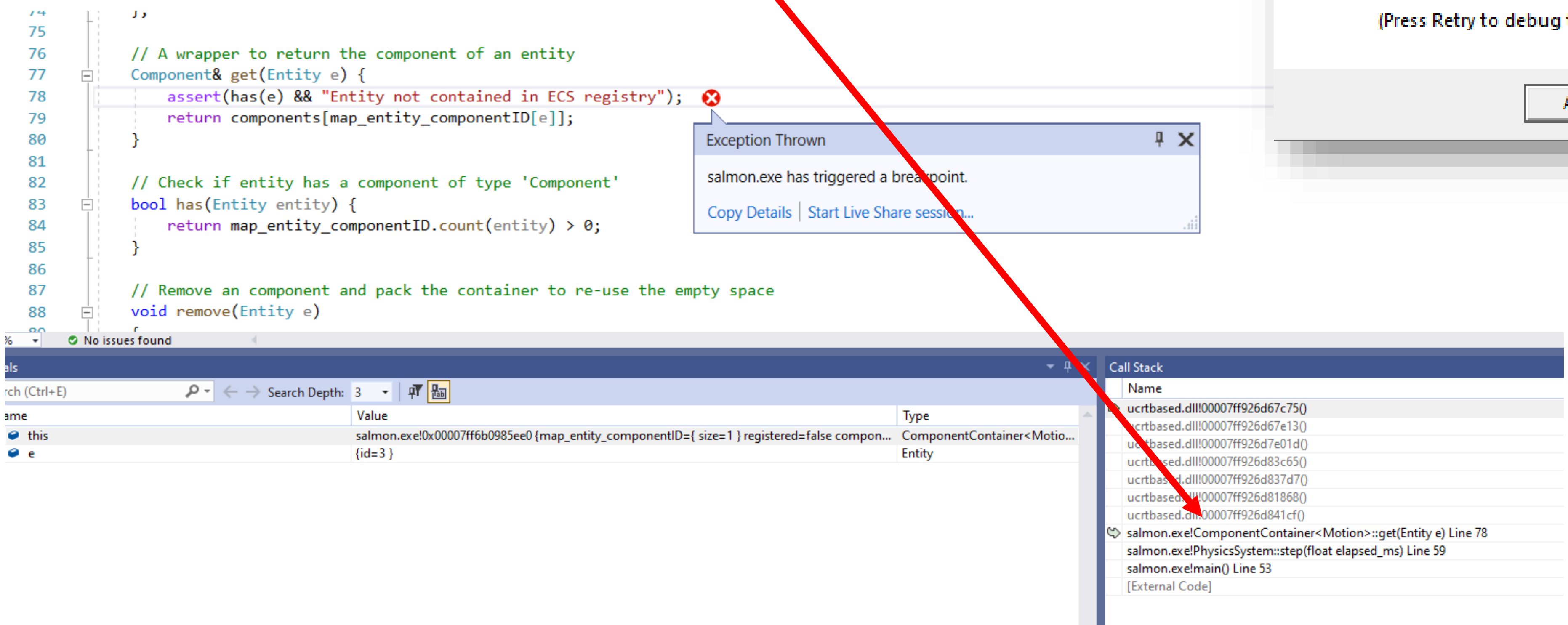
Microsoft Visual C++ Runtime Library

Debug Error!

Program:
C:\Code\cpsc-427-dev2\template\out\build\x64-Debug\salmon.exe

abort() has been called

(Press Retry to debug the application)



```
74  };
75
76      // A wrapper to return the component of an entity
77      Component& get(Entity e) {
78          assert(has(e) && "Entity not contained in ECS registry");
79          return components[map_entity_componentID[e]];
80      }
81
82      // Check if entity has a component of type 'Component'
83      bool has(Entity entity) {
84          return map_entity_componentID.count(entity) > 0;
85      }
86
87      // Remove an component and pack the container to re-use the empty space
88      void remove(Entity e)
89      {
```

Exception Thrown

salmon.exe has triggered a breakpoint.

Name	Value	Type
this	salmon.exe!0x00007ff6b0985ee0 {map_entity_componentID={ size=1 } registered=false compon...	ComponentContainer<Motio...
e	{id=3 }	Entity

Call Stack

```
Name
ucrtbased.dll!00007ff926d67c75()
ucrtbased.dll!00007ff926d67e13()
ucrtbased.dll!00007ff926d7e01d()
ucrtbased.dll!00007ff926d83c65()
ucrtbased.dll!00007ff926d837d7()
ucrtbased.dll!00007ff926d81868()
ucrtbased.dll!00007ff926d841cf()
salmon.exe!ComponentContainer<Motion>::get(Entity e) Line 78
salmon.exe!PhysicsSystem::step(float elapsed_ms) Line 59
salmon.exe!main() Line 53
[External Code]
```

Debugging

- ***There will be bugs…***

- ***Strategies for Fixing?***

Debugging

- ***There will be bugs…***

- ***Strategies for Fixing?***
  - Anticipate
  - Reproduce
  - 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*

Debugging (From Waterloo ECE 155, Zarnett & Lam)

- ***Strategies for Fixing?***
- Scientific method.
  - Observe a failure.
  - Invent a hypothesis.
  - 3 Make predictions.
  - 4 Test the predictions using experiments and observations.
- Correct? Refine the hypothesis.
- Wrong? Try again with a new hypothesis.
- Repeat

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