CPSC 427
Video Game Programming

AI Reloaded

Helge Rhodin

Overview

1. ***Recap Behaviour trees***
2. ***Shortest path and other search algorithms***

Learning outcome:

- ***Link data structure and algorithm knowledge to game dev.***
- ***Understand search algorithms (breadth first, depth first, A*, min max)***

Upcoming:

- ***Guest lecture on Wednesday (via zoom)***
- ***A2 submission on Fr.***
- *M2 submission and A2 grading the week after*

Setup

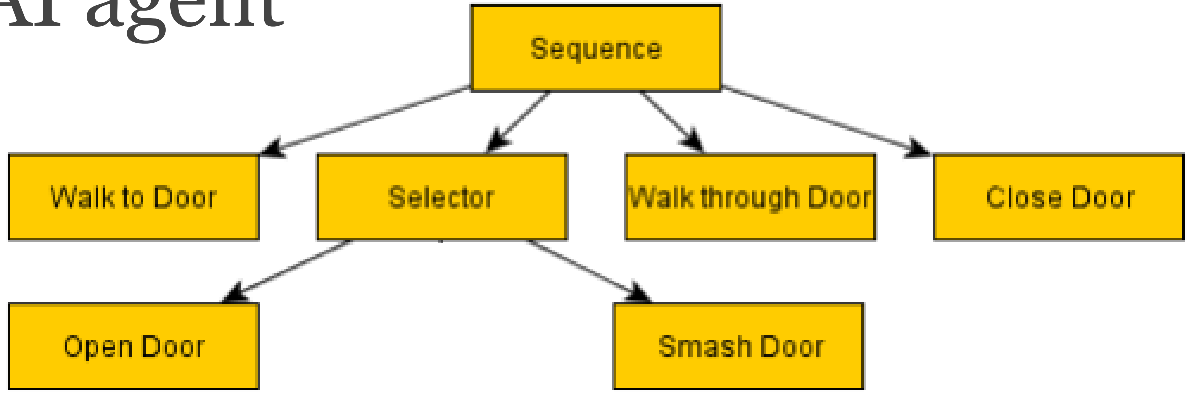
@Helge: Pressed record?

@Class: Logged into iClicker cloud?

Optional: Download and compile example BTree code

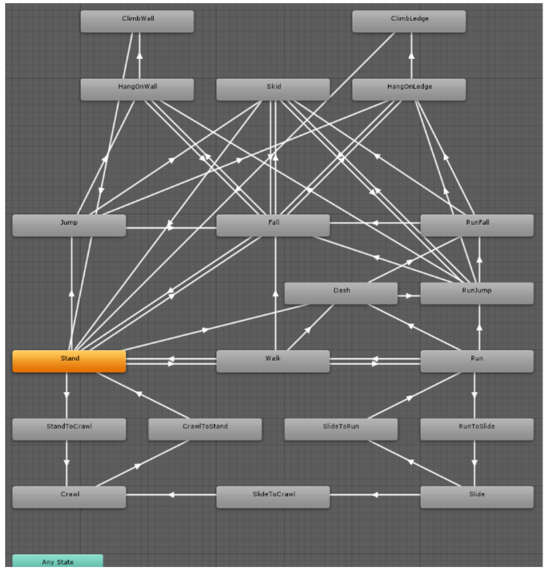
Recap: Behaviour Trees

- flow of decision making of an AI agent
- tree structured
- ***Each frame:***
  - Visit nodes from root to leaves
    - *depth-first order*
    - *check currently running node*
      - succeeds or fails:
        - return to parent node and evaluate its Success/Failure
        - the parent may call new branches in sequence or return Success/Failure
      - continues running: recursively return Running till root (usually)



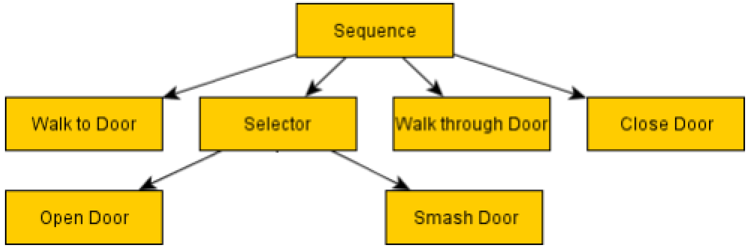
New: finite state machine vs. behaviour tree

- ***Is a behaviour tree a state machine?***
  - *State-based?*
    - in each step, one b-tree node is running
  - *Transitions?*
    - yes, with special tree structure



- Why are b-trees better?

- Can more constraints be helpful?



Recap: Leaf Nodes

Functionality

- ***init(…)***
  - *Called by parent to initialize*
  - *Sets state to Running*
  - *Not called gain before returning Success/Failure*
- ***process()***
  - *Called every frame/tick the node is running*
  - *Does internal processing, interacts with the world*
  - *Returns Running/Success/Failure*

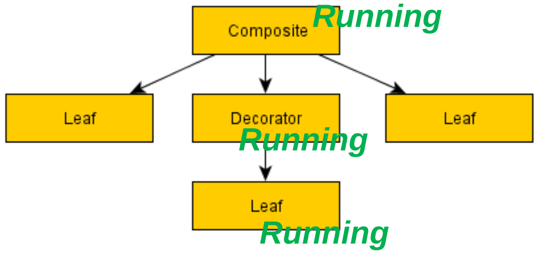
Example: *Walk to goal location*

- *Sets goal position for path finding*
- Computes shortest path
- Sets character velocity
- Returns
  - success: Reached destination
  - failure: No path found
  - running: En route

Recap: Early exit?

- ***All parents of the currently running leaf node are running too***
- ***A node early in the tree can return Success/Failure***
  - Terminates children implicitly

Example



- ***upon alarm***
  - abort sleeping
  - init walking node

- **Trying again?**
  - Re-initialize children with new parameters to init(…)

- ***try to sleep if alarm is off***
  - init sleeping node

Recap: Implementation example

Basics:

```
// The return type of behaviour tree processing
enum class BTState {
    Running,
    Success,
    Failure
};


// The base class representing any node in our behaviour tree
class BTNode {
public:
    virtual void init(Entity e) {};

    virtual BTState process(Entity e) = 0;
};
```

if condition (inflexible)

```
// A general decorator with lambda condition
class BTIfCondition : public BTNode
{
public:
    BTIfCondition(BTNode* child)
        : m_child(child) {
    }

    virtual void init(Entity e) override {
        m_child->init(e);
    }

    virtual BTState process(Entity e) override {
        if (registry.motions.has(e)) // hardocded
            return m_child->process(e);
        else
            return BTState::Success;
    }
private:
    BTNode* m_child;
};
```

Recap: Implementation example II

A leaf node

```
class TurnAround : public BTNode {
private:
    void init(Entity e) override {
    }

    BTState process(Entity e) override {
        // modify world
        auto& vel = registry.motions.get(e).velocity;
        vel = -vel;

        // return progress
        return BTState::Success;
    }
};
```

New: A leaf node with internal state

Example scenarios

1. ***Run three steps, turn around, run one step back***

2. ***Turn right, run three steps, turn around***

Live demo

Multiple components for one entity?

Classical ECS:

- ***Each entity***
  - has one ID
  - has or has not a certain component type
  - cannot store multiple components of the same type

Character inventory:

- A character should be able to hold multiple portions of the same type
- Solution:
  - Each item is its own entity
  - Introduce an inventory component that stores list of items (list of entities)

The same b-tree for multiple entities?

- ***How to store the state with each entity?***
  - within the ECS registry?
    - *add a new state component for each b-tree node?*
      - what if multiple nodes of the same type run on the same entities?
  - a custom data structure?
    - *a lookup table?*
      - conditioned on entity ID!

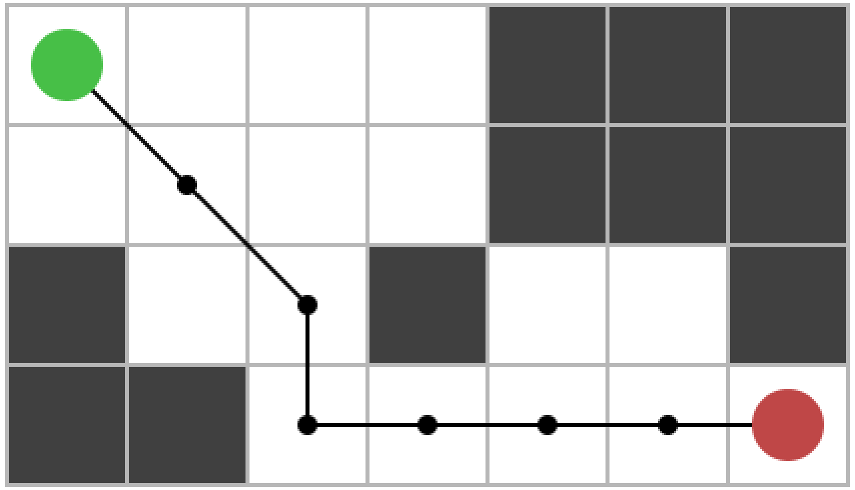
Strategy

- Given current state, determine **BEST** next move
- Short term: best among immediate options
- Long term: what brings something closest to a goal
  - *How?*
    - Search for path to best outcome
      - Across states/state parameters

ivanhoe.pro

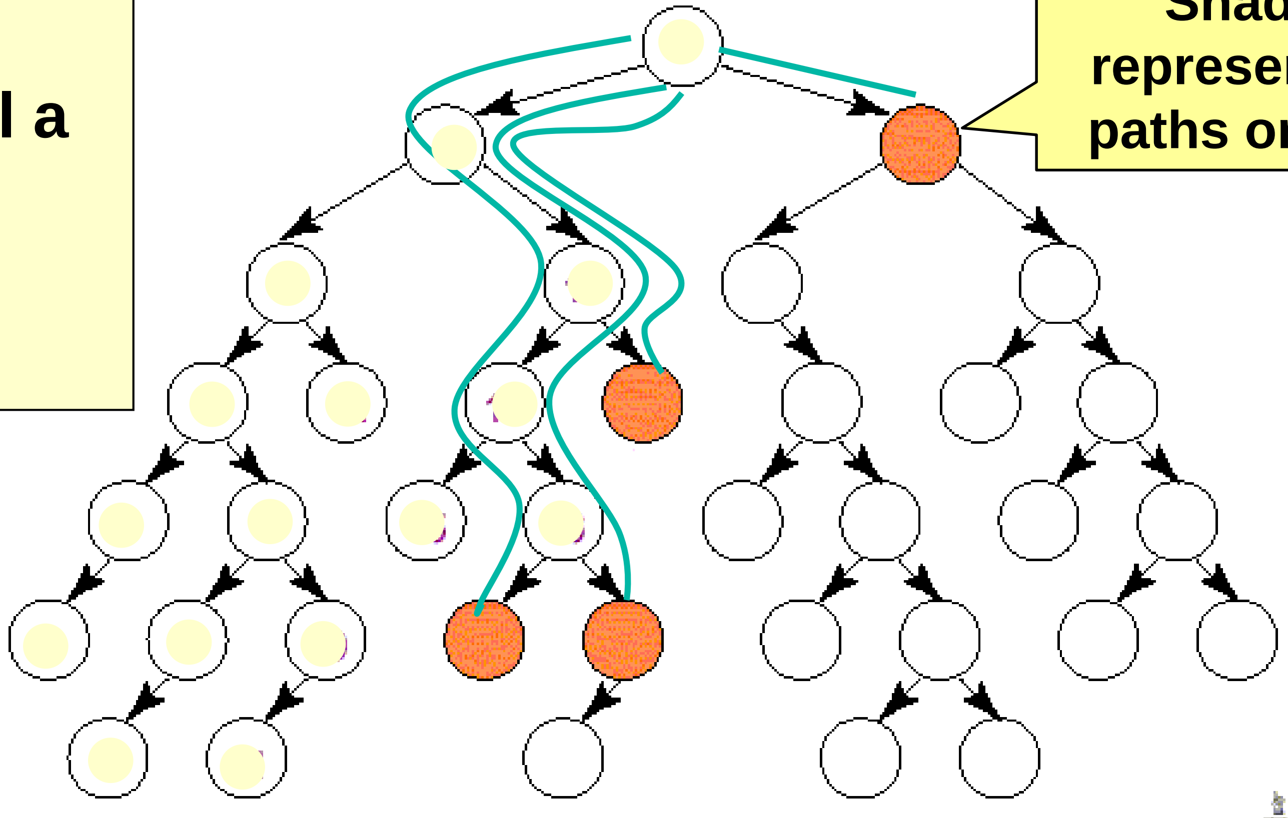
Pathfinding

- **How do I get from point A to point B?**



DFS: Depth-first search

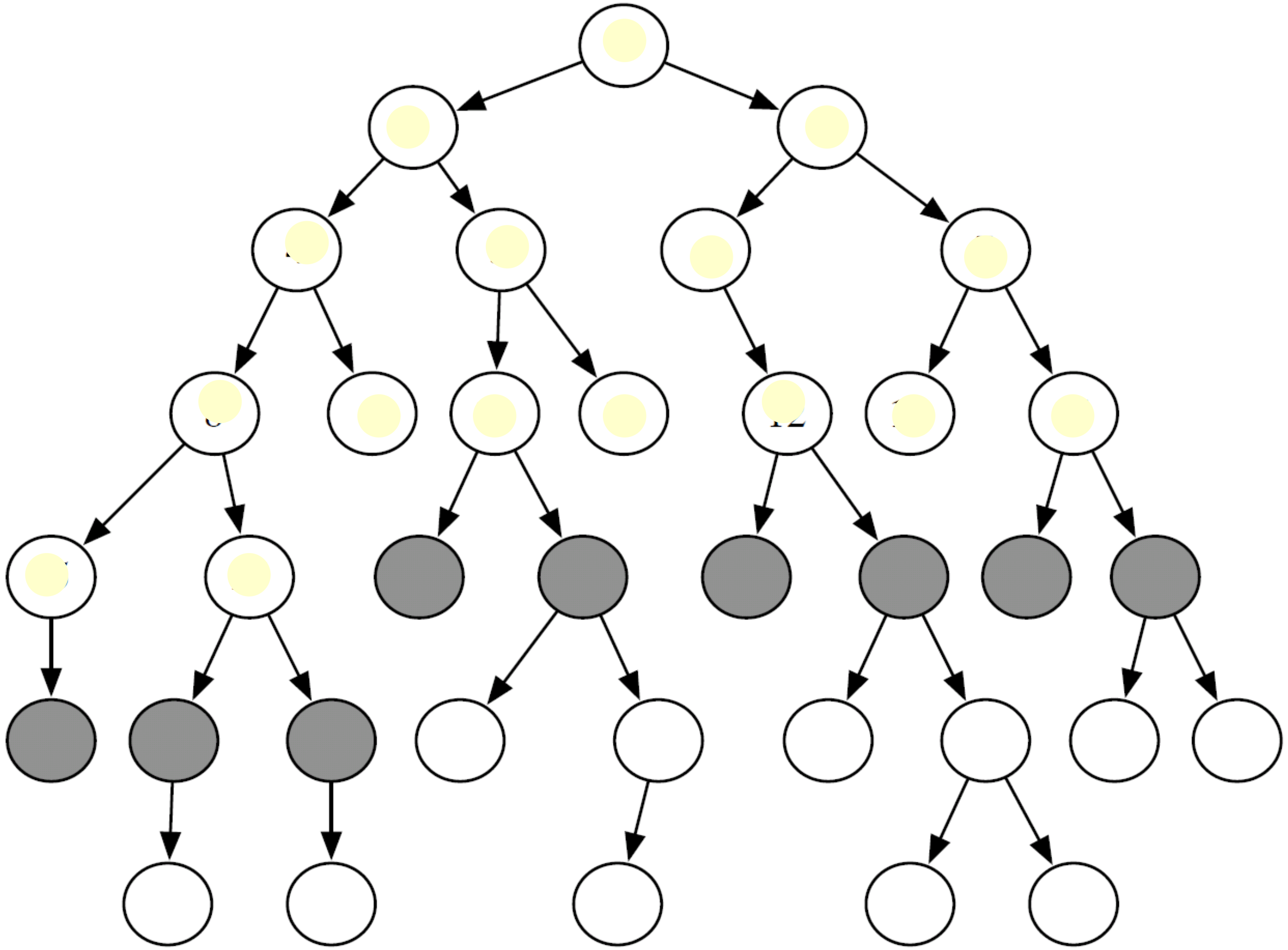
Explore each path on the frontier until its end (or until a goal is found) before considering any other path.



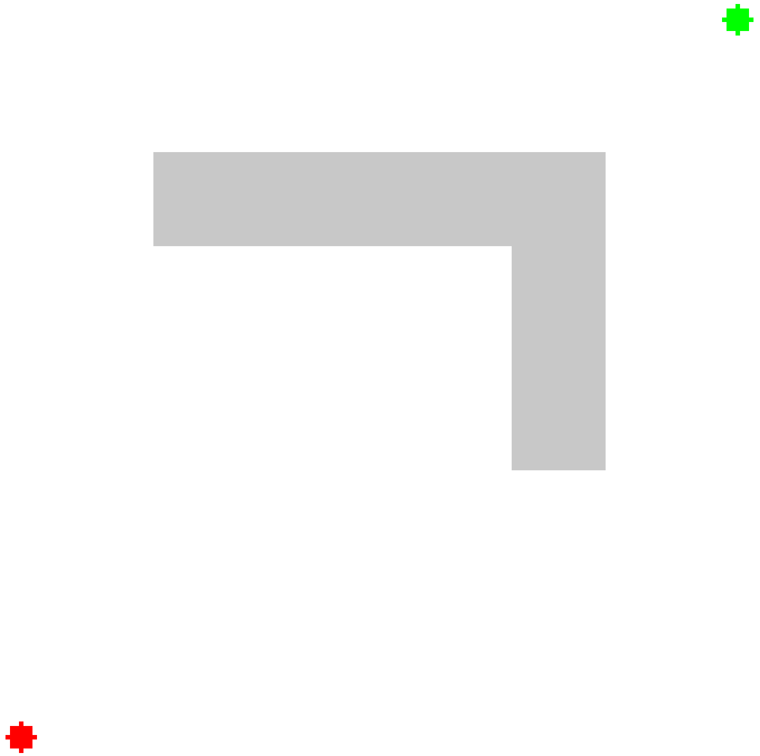
Shaded nodes represent the end of paths on the frontier

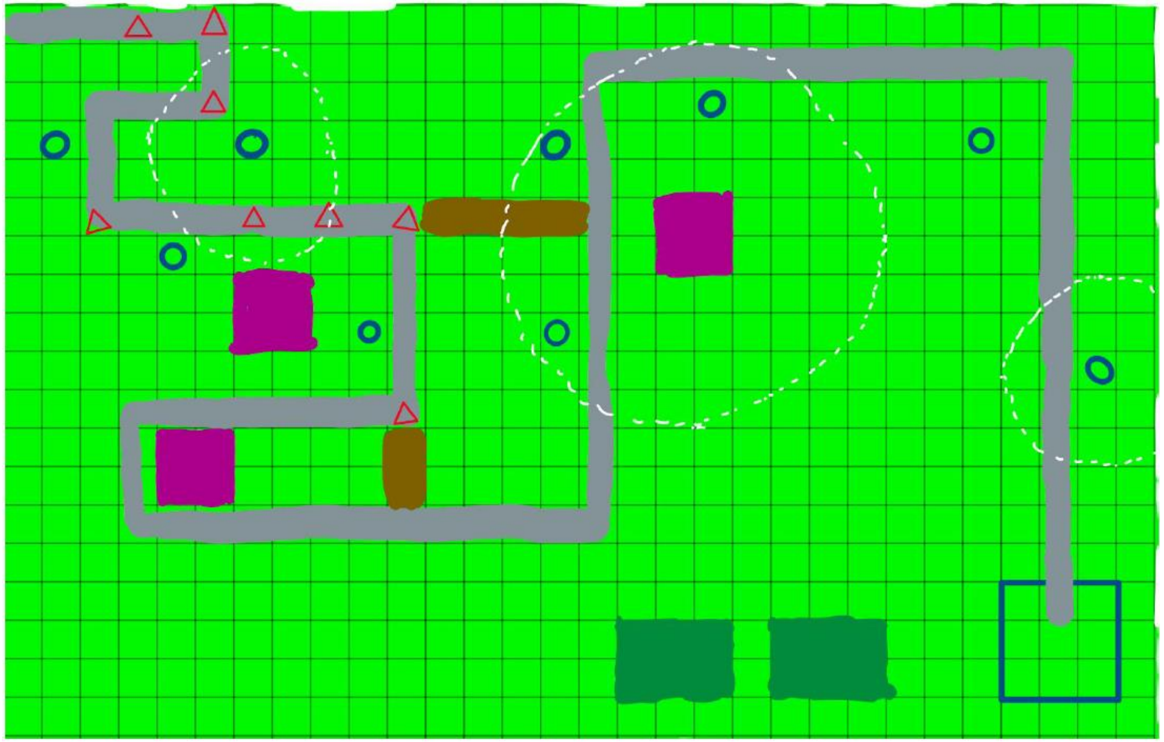
Breadth-first search (BFS)

- **Explore all paths of length L on the frontier, before looking at path of length *L + 1***



Breadth-first





Project pitch Team 4

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

When to use BFS vs. DFS?

- ***The search graph has cycles or is infinite***

  BFS

- ***We need the shortest path to a solution***

  BFS

- ***There are only solutions at great depth***

  DFS

- ***There are some solutions at shallow depth***

  BFS

- ***No way the search graph will fit into memory***

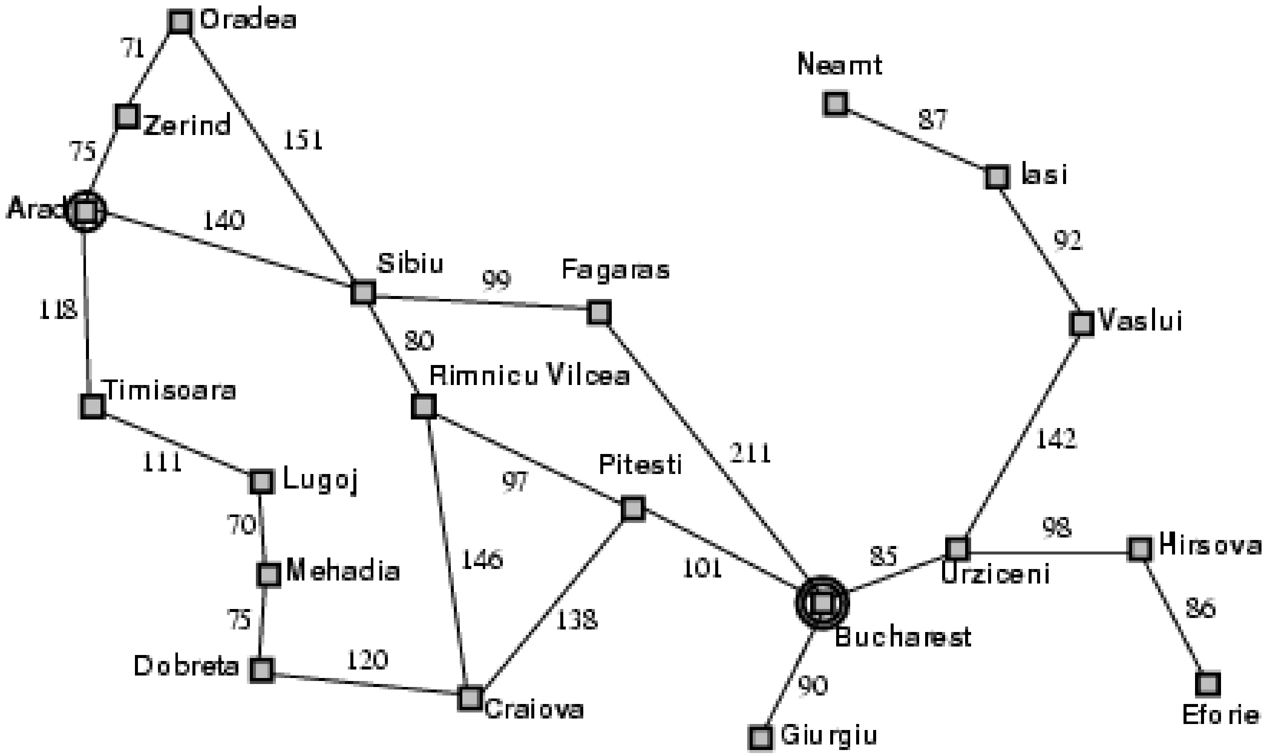
  DFS

Search with Costs

Def.: The cost of a path is the sum of the costs of its arcs

$$\text{cost}(\langle n_0, \ldots, n_k \rangle) = \sum_{i=1}^{k} \text{cost}(\langle n_{i-1}, n_i \rangle)$$

Want to find the solution that minimizes cost



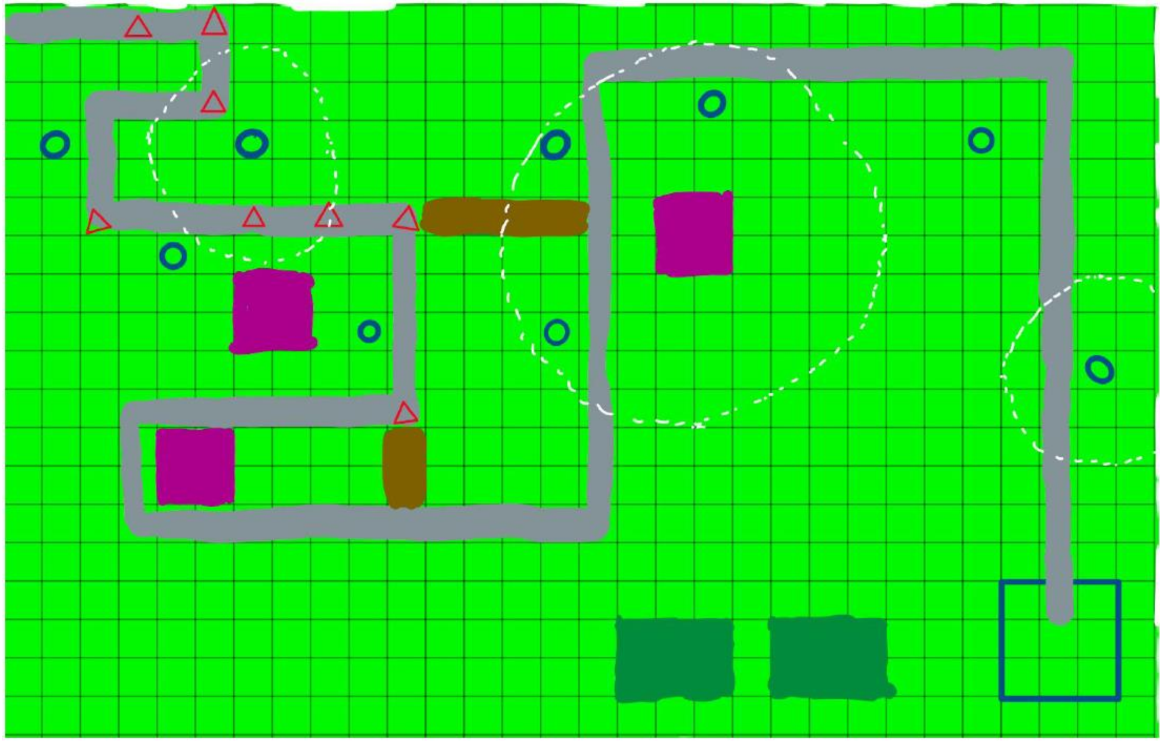
Example: Tower Defence

Normal unit motion cost:

- Street: cost 1
- Other: cost infinity

Boss unit: *which shortcuts will it take?*

- Street: cost 1
- Dirt road: cost 5
- Grass: cost 50
- Purple stuff: cost infinity



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

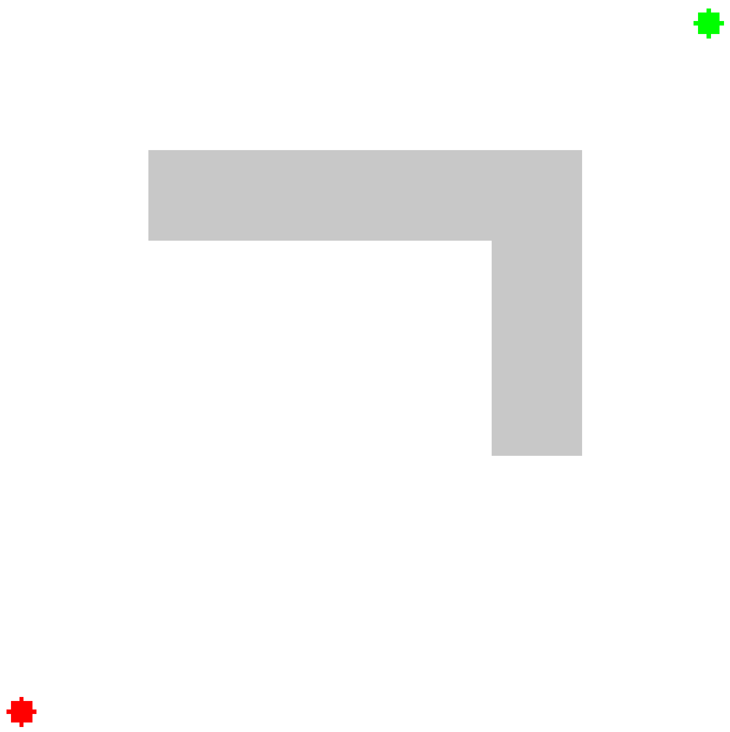
- We often have estimates of distance/cost from node n to a goal node

- **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*

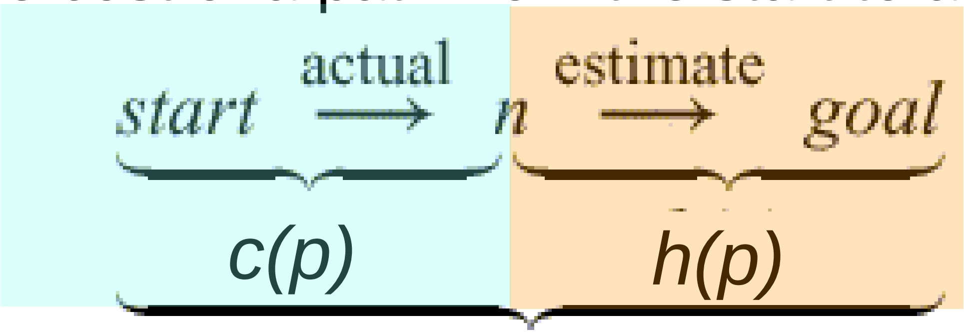
A* search



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* implementation

- 1. Initialize open and closed lists.
  - Put starting node on open list.
- 2. While open list is not empty:
  - Find node with smallest f on the list, call it q
  - Pop q off of open list
  - Find q's "successors", and set their parent nodes to q

A* implementation

- 1. Initialize open, closed lists. Put starting node on open list.
- 2. While open list is not empty:
  - Find node with smallest f on the list, call it q
  - Pop q off of open list
  - Find q's "successors", and set their parent nodes to q
  - For each successor:
    - If successor is the goal, done!
    - c(successor) = c(q) + d(q,successor)
      h(successor) = D(goal, successor)
    - If successor already exists in open list with lower f = c + h, skip it
    - If successor already exists in closed list with lower f, skip it
    - Otherwise, add successor to open list

A* implementation

- 1. Initialize open, closed lists. Put starting node on open list.
- 2. While open list is not empty:
  - Find node with smallest f on the list, call it q
  - Pop q off of open list
  - Find q's "successors", and set their parent nodes to q
  - For each successor:
    - If successor is the goal, done!
    - g(successor) = g(q) + d(q,successor)
      h(successor) = d(goal, successor)
    - If successor already exists in open list with lower f, skip it
    - If successor already exists in closed list with lower f, skip it
    - Otherwise, add successor to open list
  - **Put q on closed list**

A* search

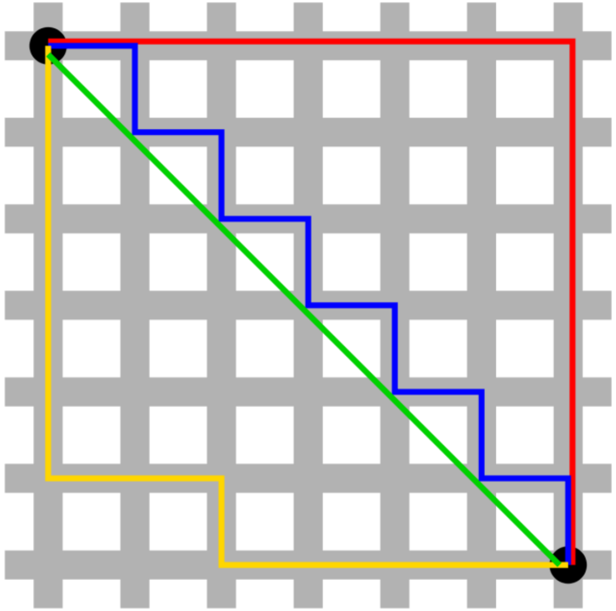
Key idea: H is a heuristic, and not the real distance:

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

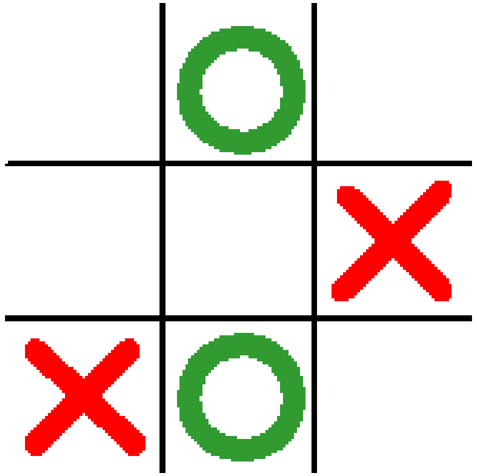
www.npr.org

Min-Max Trees

- Adversarial planning in a turn-taking environment
  - *Algorithm seeks to maximize our success* ***F***
  - *Adversary seeks to minimize* ***F***
  - $\boldsymbol{a_{we}} = \max_{\boldsymbol{we}} \min_{they} \boldsymbol{F}(\boldsymbol{a_{we}}, \boldsymbol{a_{they}})$
- 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*

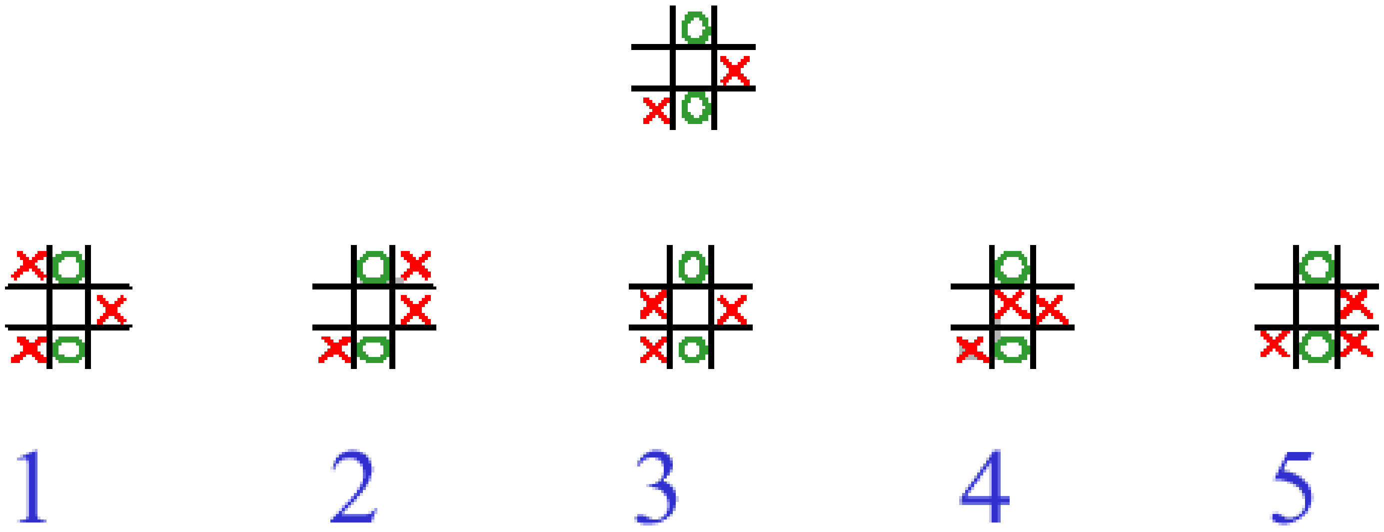
Example

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



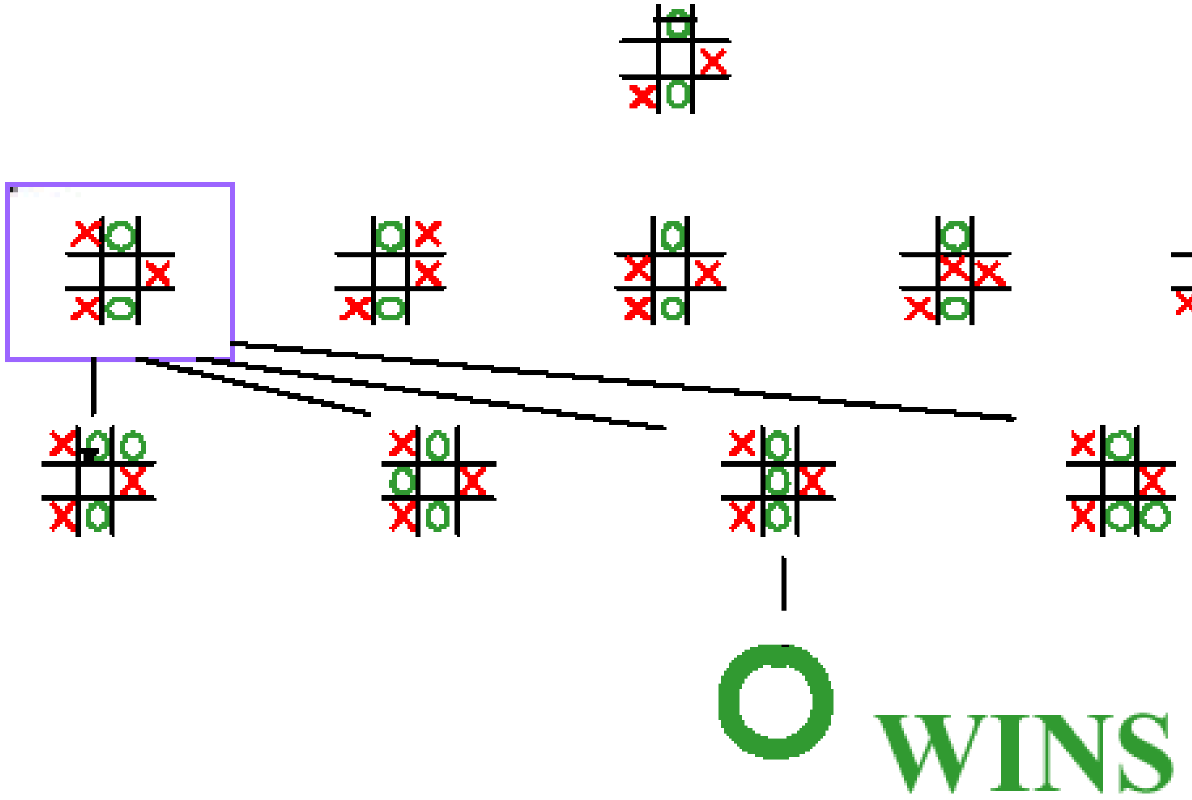
We are playing X, and it is now our turn.

Our options:



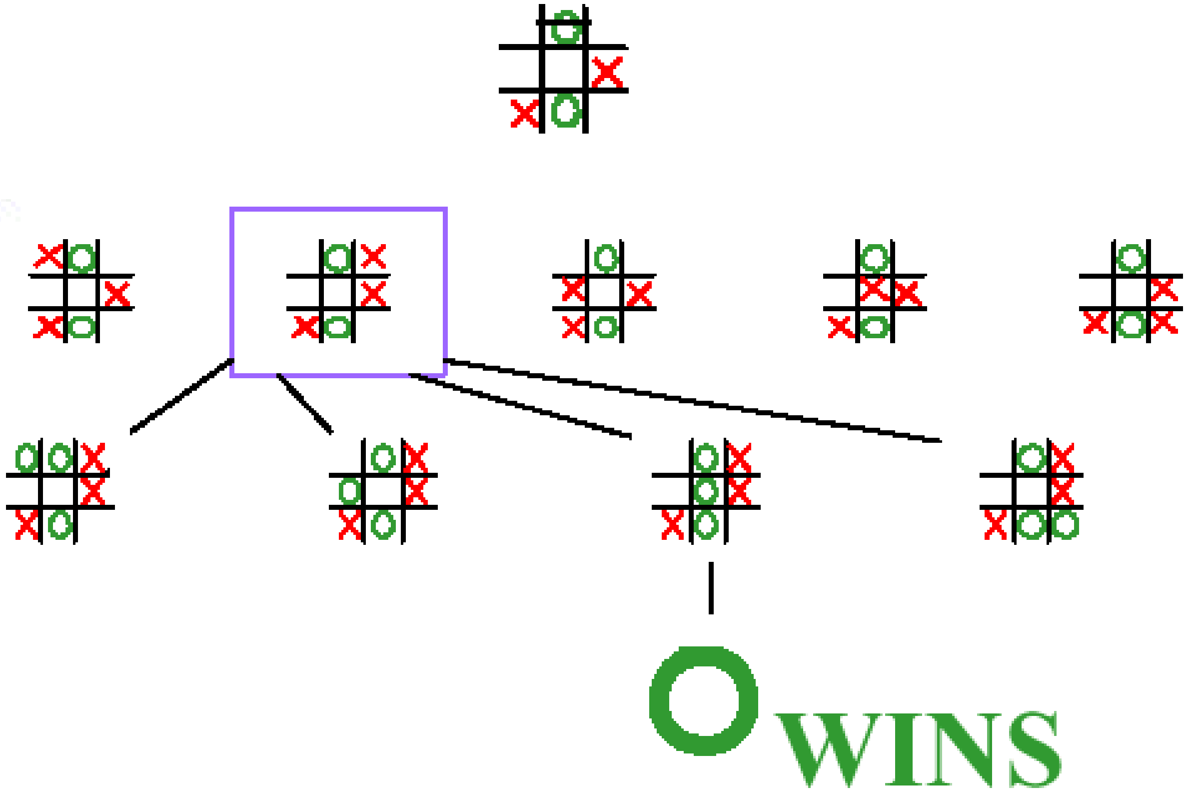
Number = position after each legal move

Opponent options



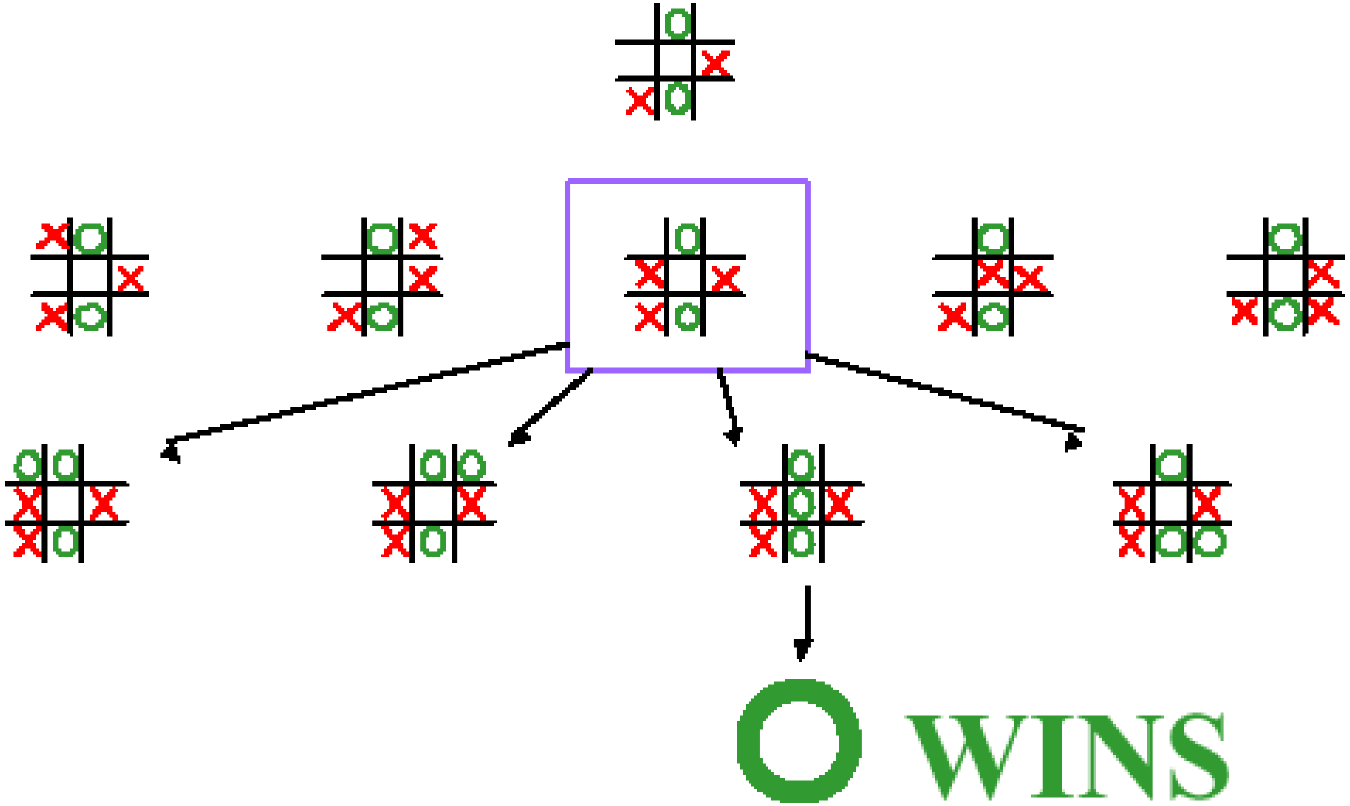
Here we are looking at all of the opponent responses to the first possible move we could make.

Opponent options

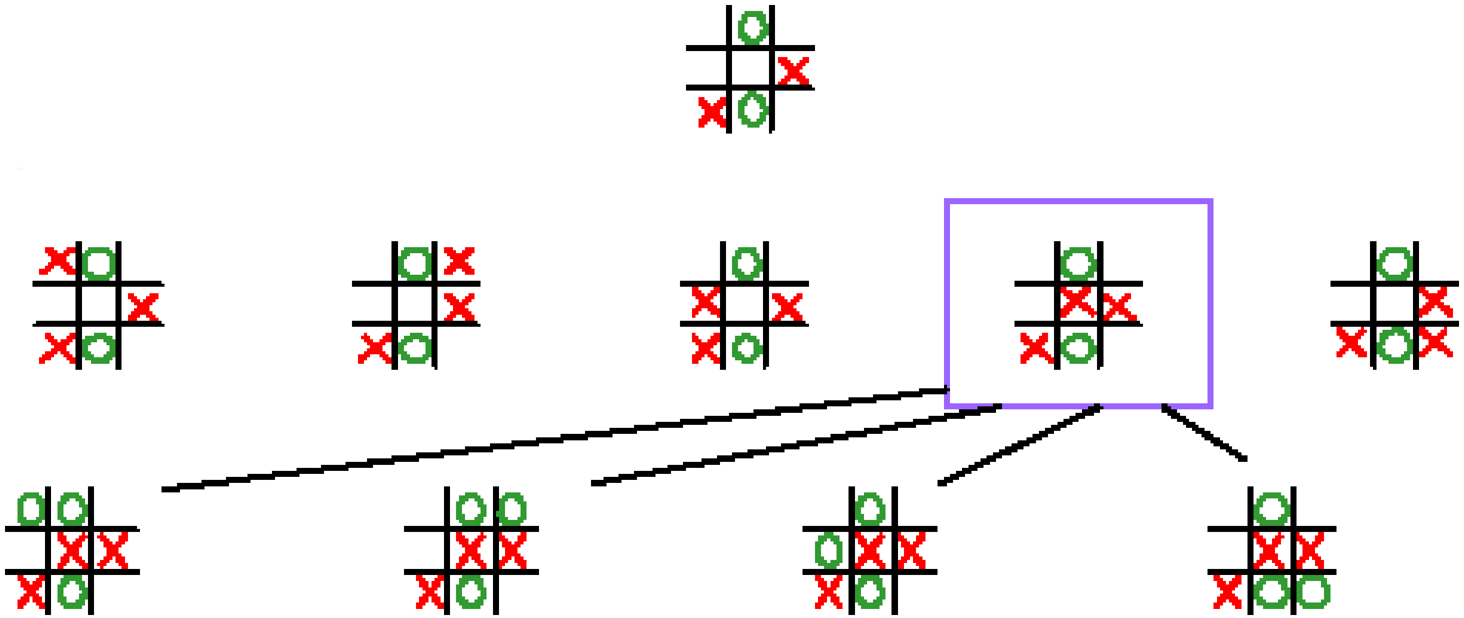


Opponent options after our second possibility. Not good again…

Opponent options

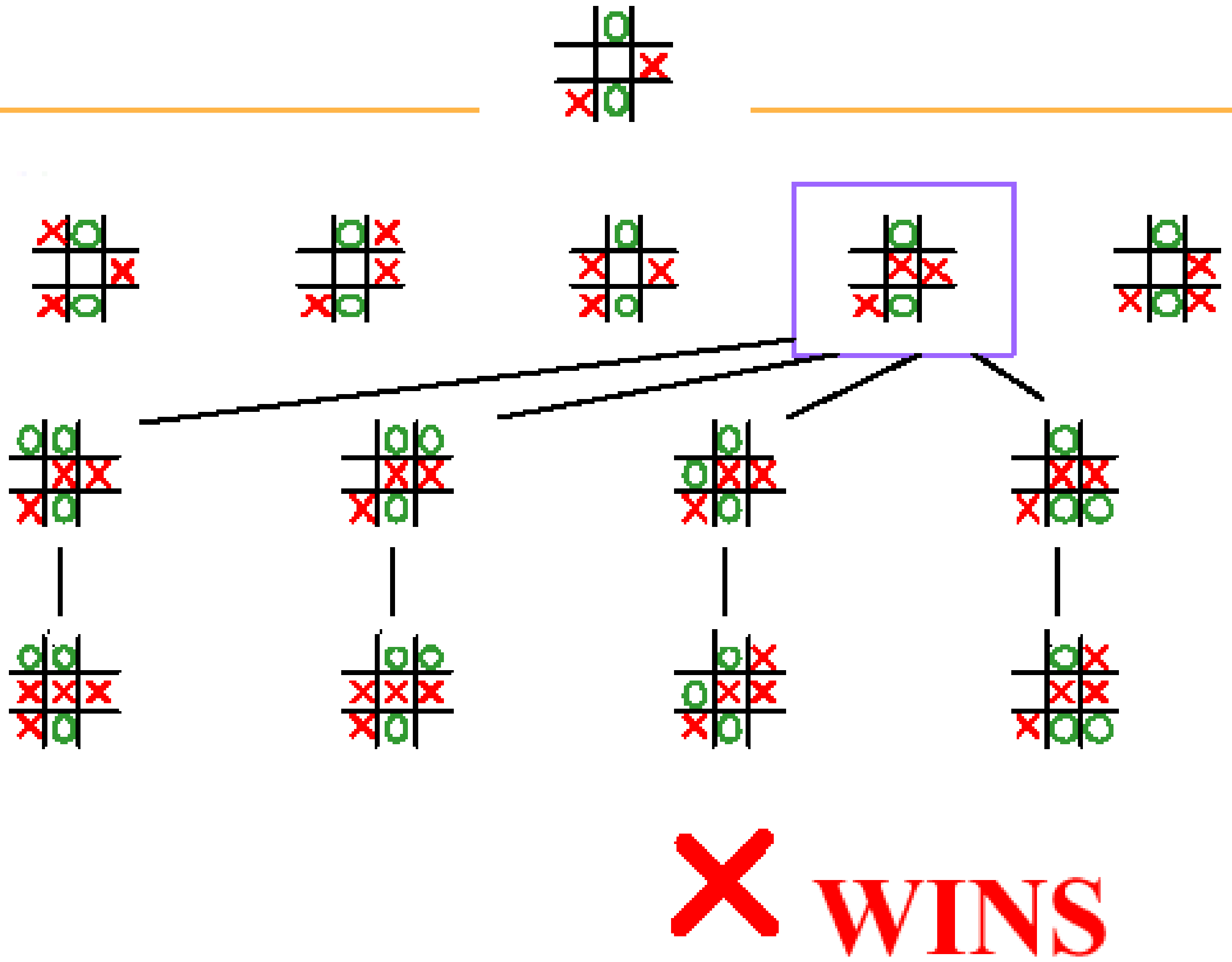


Opponent options => Our options



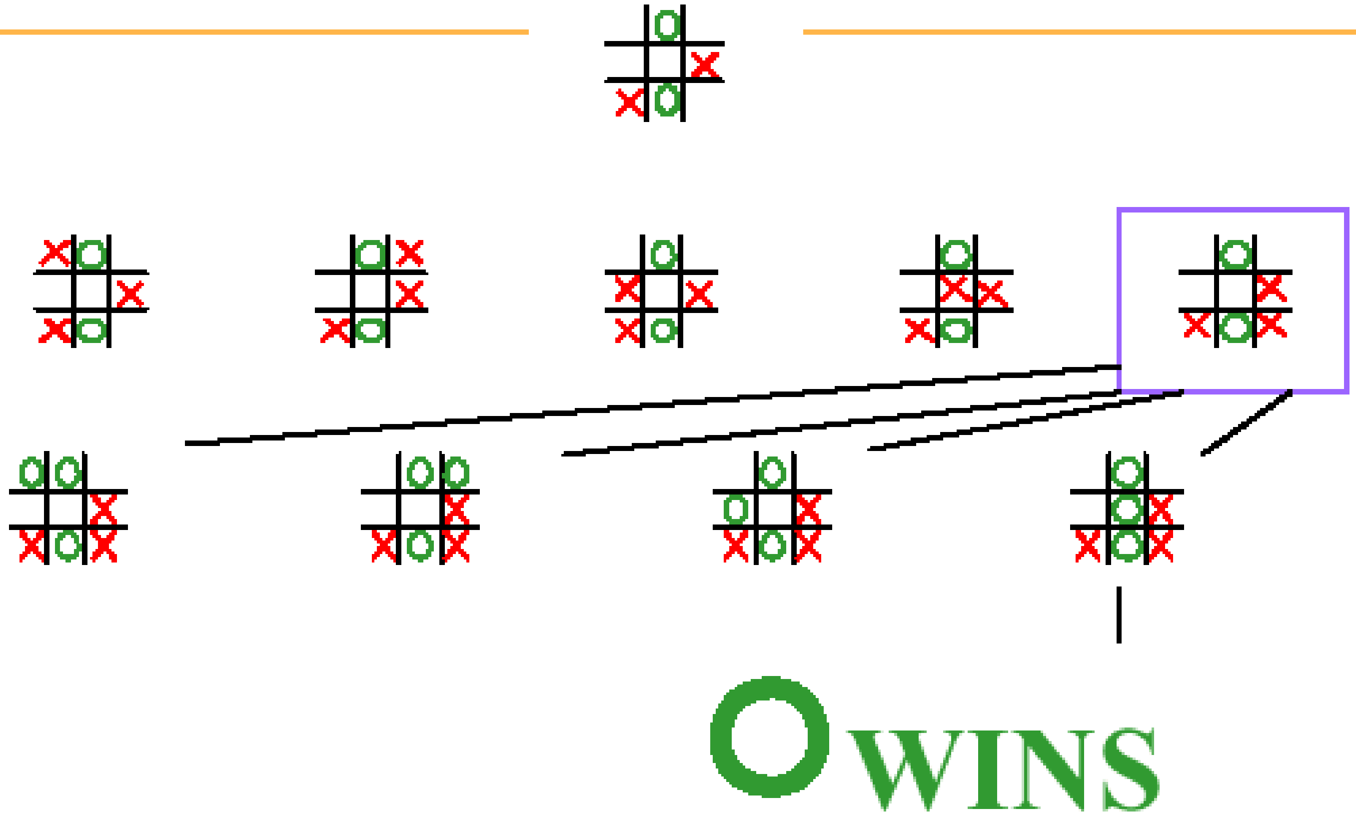
Now they don't have a way to win on their next move. So now we have to consider our responses to their responses.

Our options



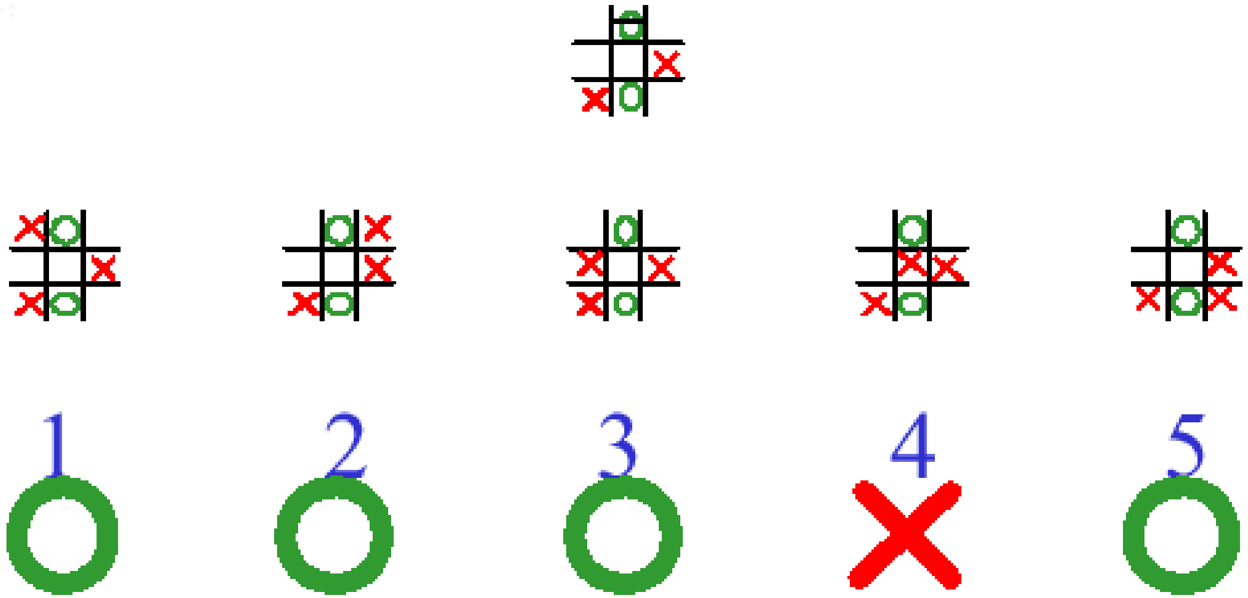
We have a win for any move they make.
Original position in purple is an X win.

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 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*
  - *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

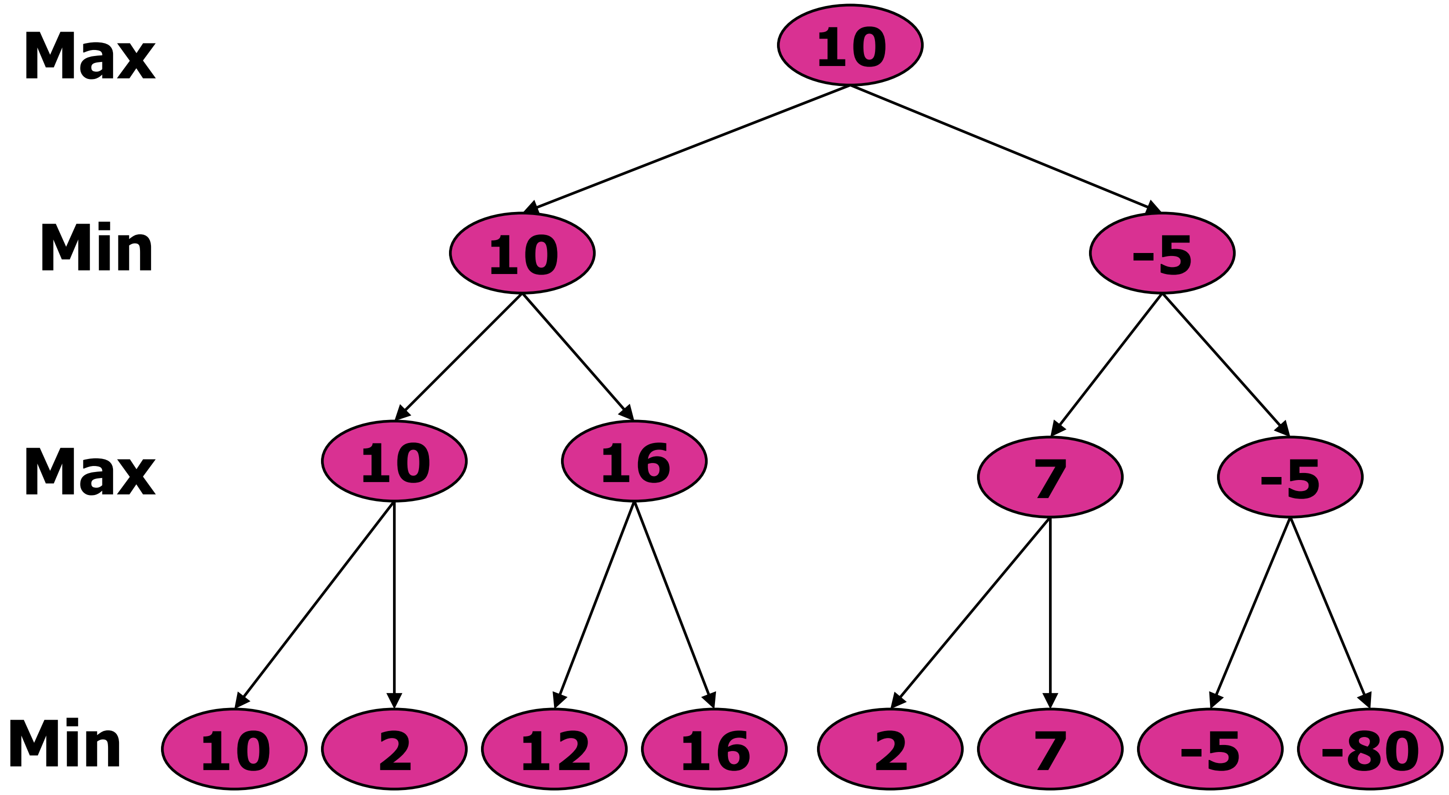
- Strategy that will let us cut off the game tree at fixed depth (layer)
- Apply heuristic scoring to bottom layer
  - *instead of just Win, Loss, Tie, we have a score.*
- For “our” level of the tree we want the move that yields the node (position) with highest score. For a “them” level “they” want the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

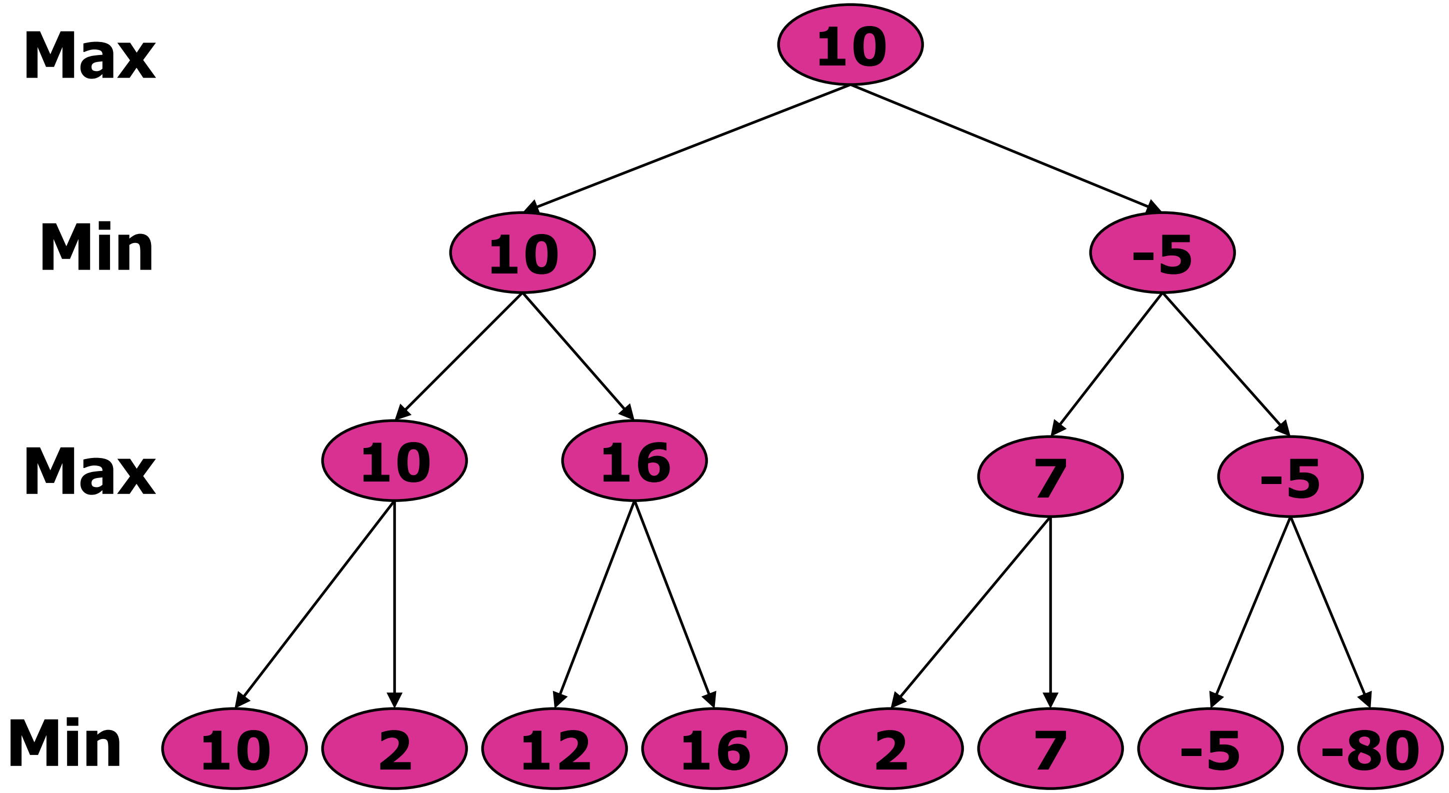
Note: we don’t use an explicit tree structure.
However, the pattern of recursive calls forms a tree on the call stack.

Real Minimax Example



Evaluation function applied to the leaves!

Pruning



Alpha Beta Pruning

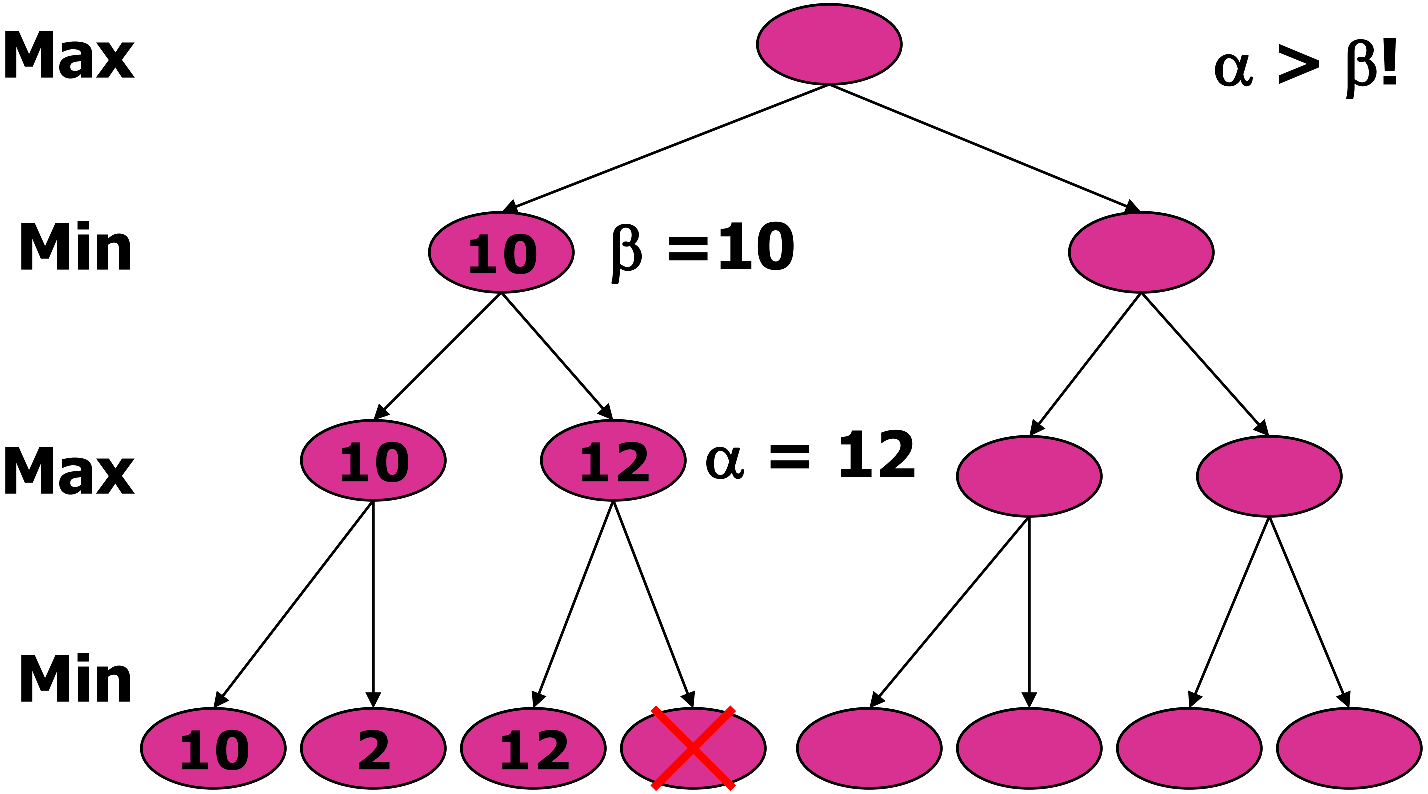
Idea: Track "window" of expectations.

Use two variables

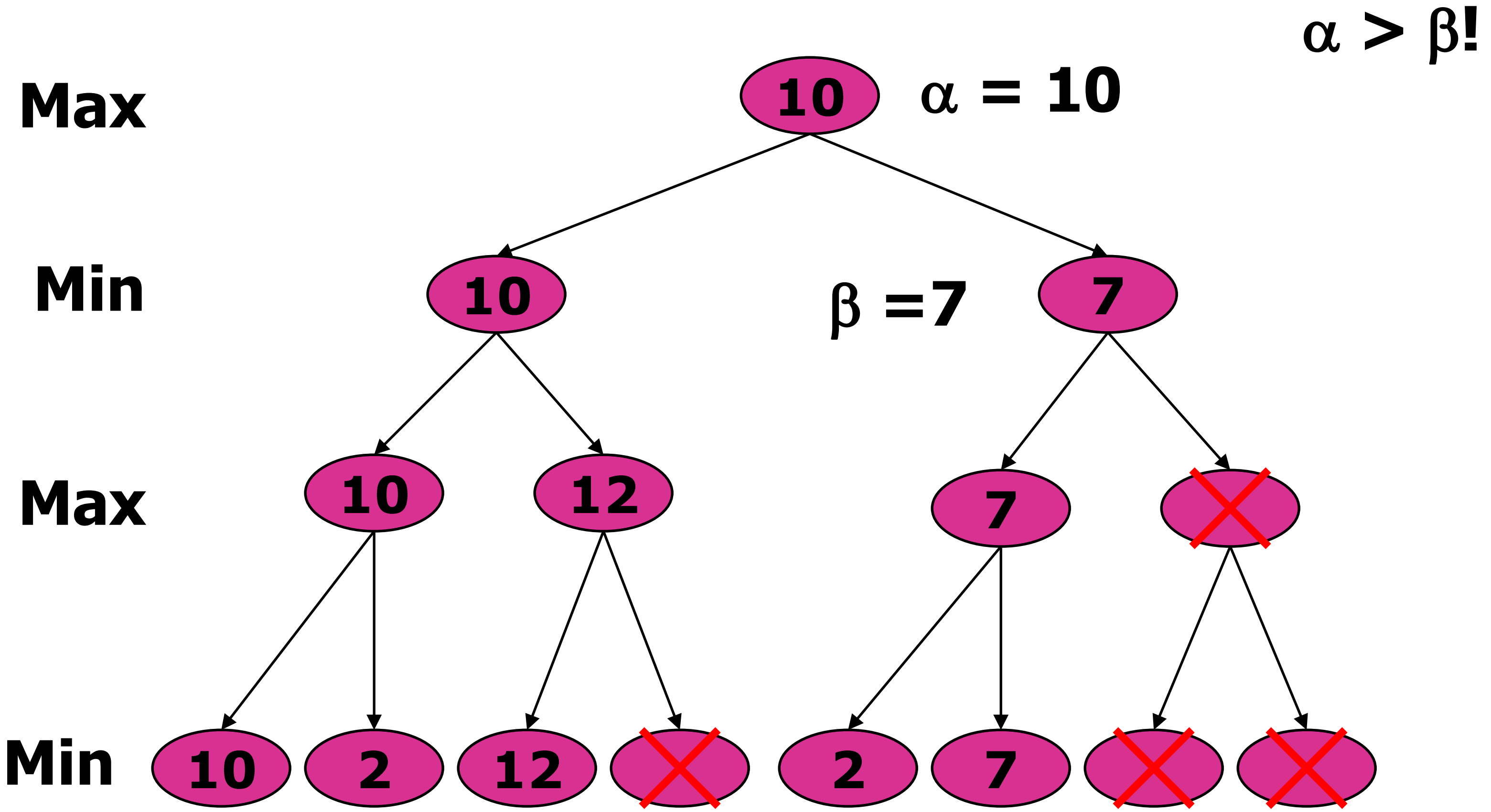
- α – Best score so far at a **max** node ('our choice'): increases
  - *At a child **min** node:*
    - Parent wants **max**. To affect the parent's current α, our β cannot drop below α.
  - *If β ever gets less:*
    - Stop searching further subtrees *of that child*. They do not matter!
- β – Best 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 ($\alpha = -\infty$, $\beta = \infty$)

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;
   }
}
```

Variants

- ***More than two players?***
- ***More than two choices?***
- ***Opponent does not select best move?***

Debugging

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