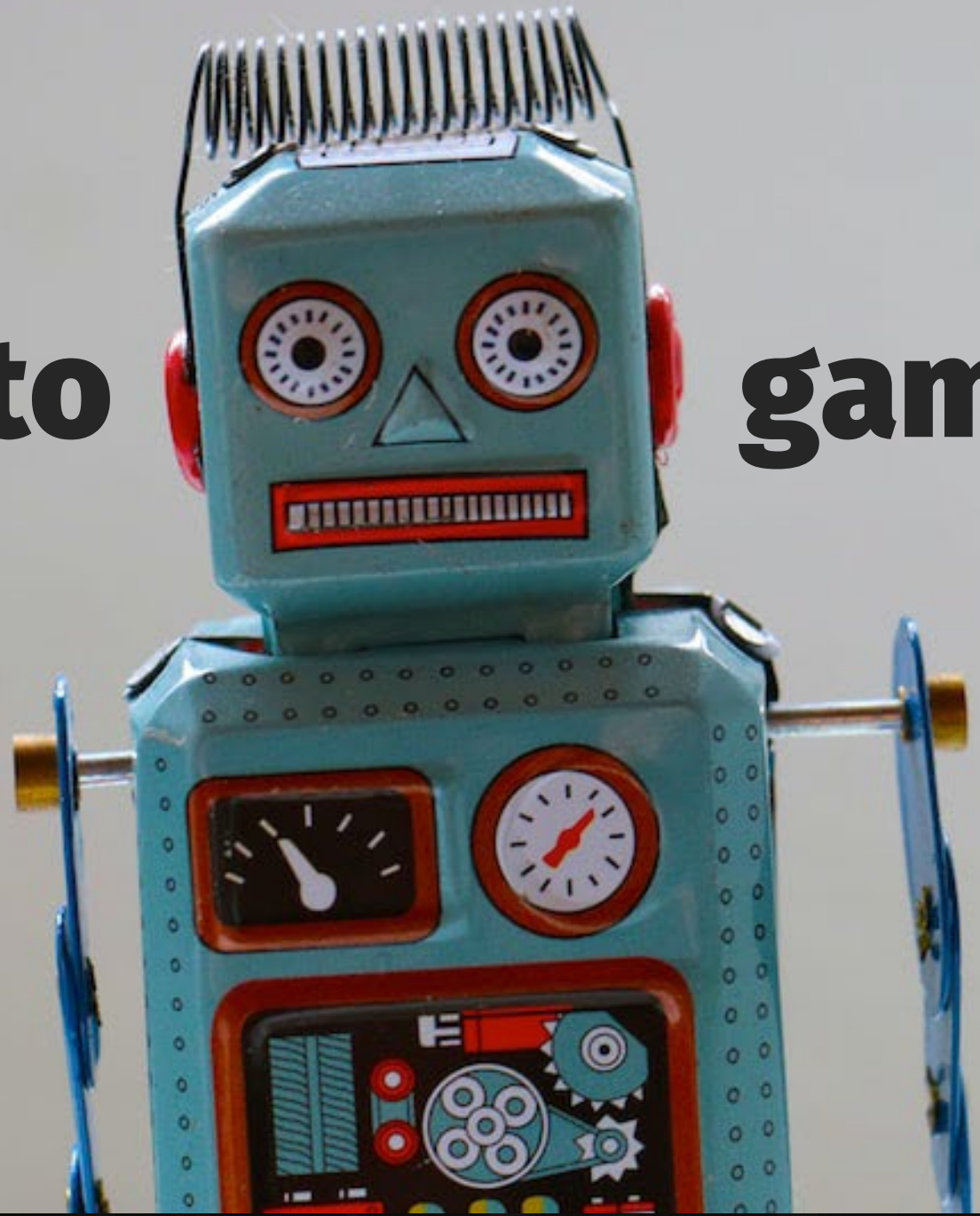
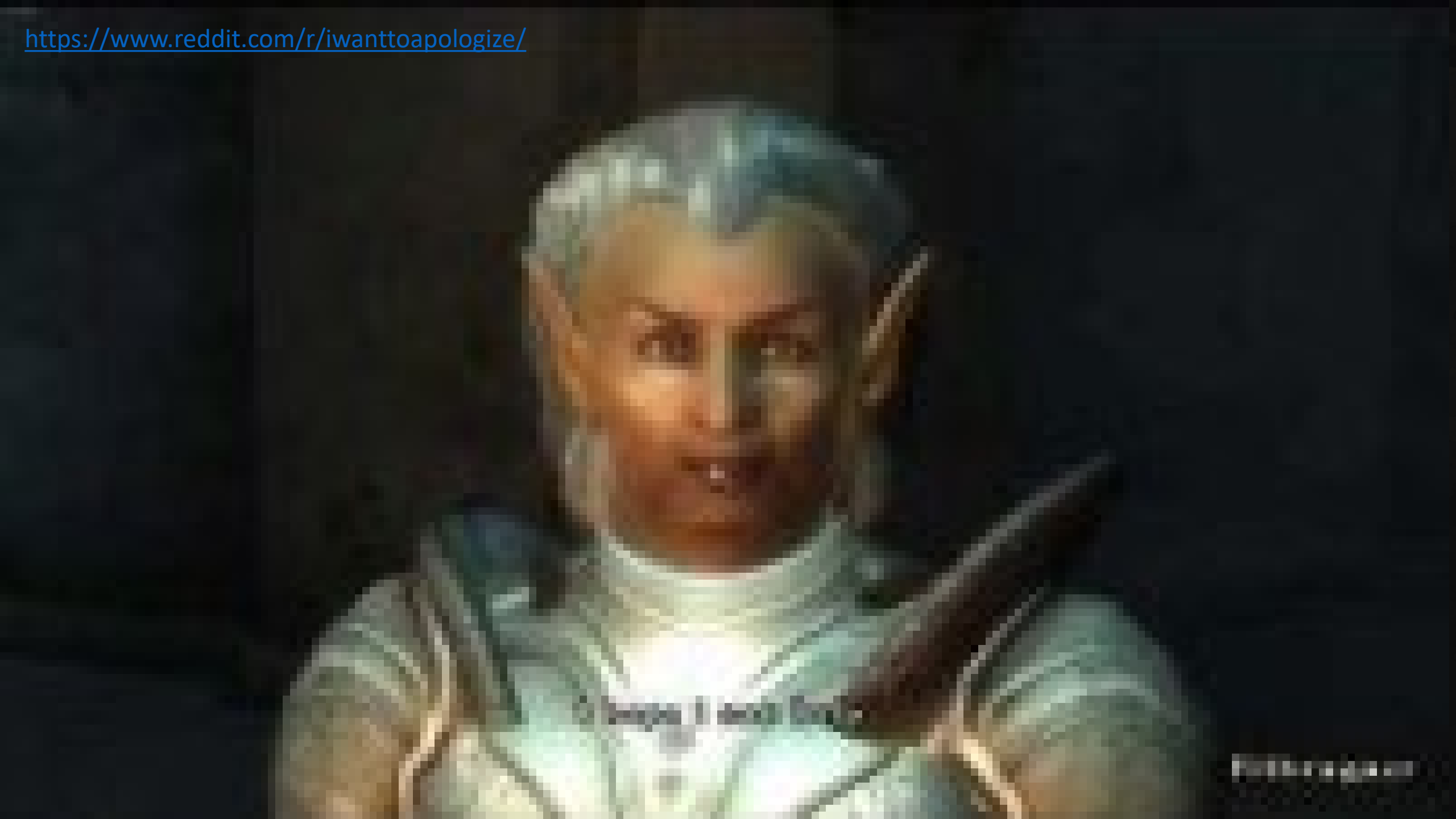
intro to game AI

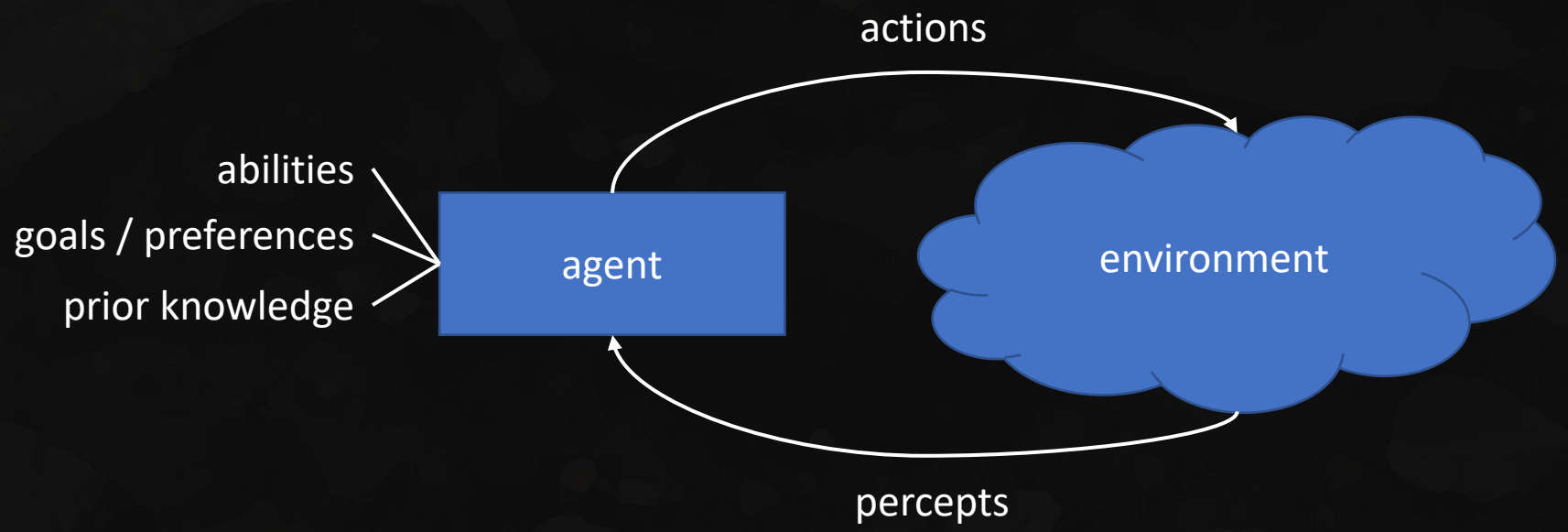


https://www.reddit.com/r/iwanttoapologize/



what is AI?

- the study of computational agents that exist in an environment and perceive and act



AI in games

- **opponents that are challenging or allies that are helpful**
- **must perform in real time**
- **configurable by designers**
  - not hard coded by programmers
- **“amount” and type of AI for game can vary**
  - RTS needs global strategy
  - FPS needs modeling of individual units at “footstep” level

game agents

- **most AI focuses on game agents**
  - enemy, ally, or neutral
- **loops through *sense-think-act* cycle**
  - acting is event specific

game agents: sensing

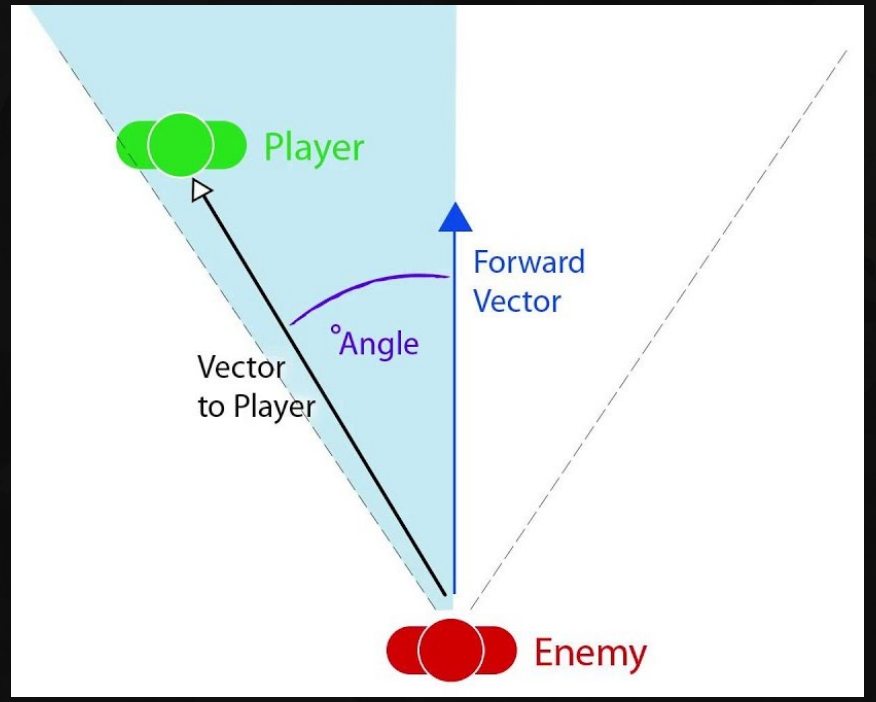
- **gather current world state: barriers, opponents, objects, etc.**
  - perceive the world and read game status
- **needs limitations**
  - avoid “cheating” by looking at game data
  - same constraints as player (vision, hearing range, etc.)

sensing types

- vision
- hearing
- communication

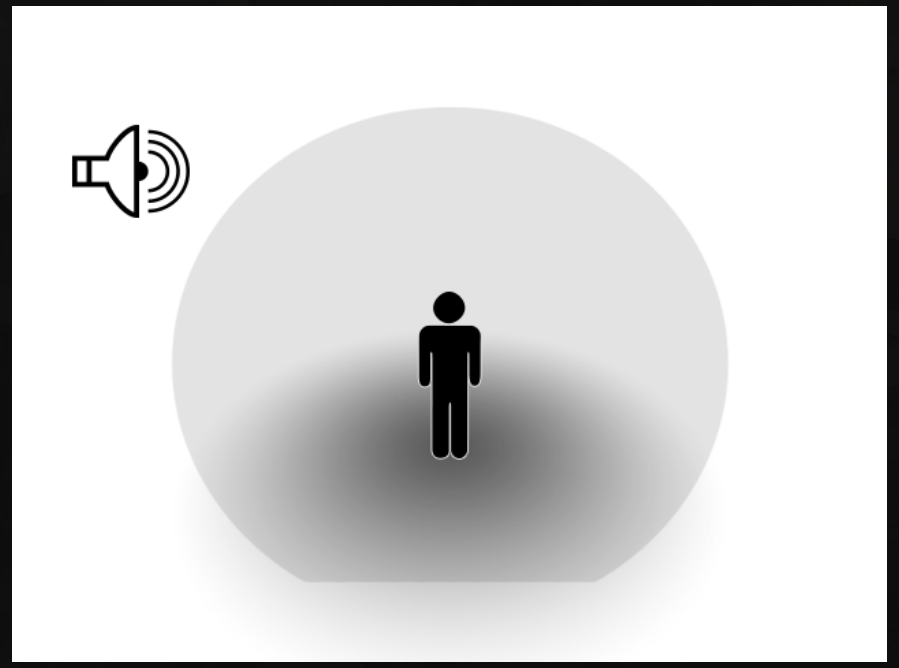
vision

- **compute vector to each object**
  - check magnitude (i.e., is it too far away?)
  - check angle (dot product) (i.e., within 120° viewing angle?)
  - check for obstacles



hearing

- **stealth mechanics**
  - tip-toe vs run
- **implement as event-driven**
  - when player performs action, notify agents withing range
  - can enhance with listen attributes by agents (if agent is “keen eared” or paying attention)



communication

- model sensing data from other agents
- can be instant (i.e., connect by radio)
- or via hearing (i.e., shout, alarm)

game agents: thinking

- **evaluate information and make decisions**
- **as simple or elaborate as required**
- **generally, three ways:**
  - pre-coded expert knowledge
  - search algorithm for best (optimal) solution
  - machine learning to adapt to player behavior

expert knowledge

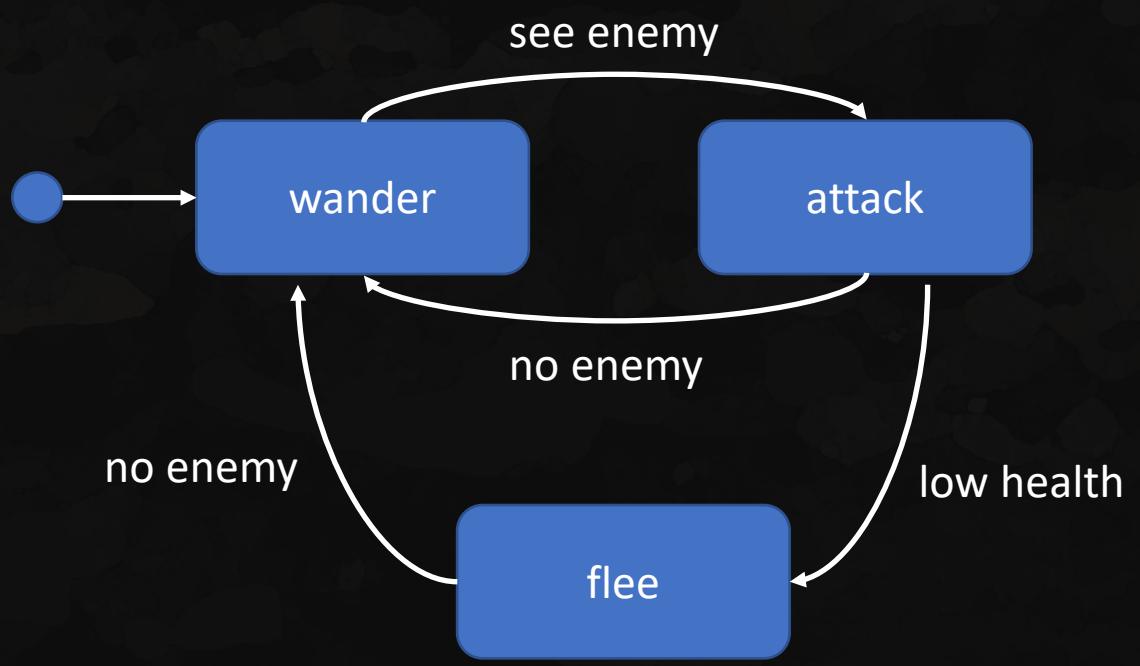
- **represent using finite state machines, decision trees, etc.**
- **common sense and knowledge of domain**
  - see enemy weaker than you → attack
  - see enemy stronger → get help

AI algorithms

- **finite state machines**
- **search**
  - permuting all possible states (e.g., a game board) to attempt to predict outcome
  - used in chess and pathfinding
- **machine learning**
  - given a set of knowledge (training), find out the wining strategy

finite state machines in AI

- abstract model of expert knowledge
- act: states
- sense: inputs
- think: transitions



search

- **permuting all possible game states to attempt to predict outcome**
- **searching for the optimal move to make given the current game state**
- **stop searching when needed**
  - you cannot always search to the end game (i.e., too many possibilities) or to the end goal (e.g., the goal is moving)

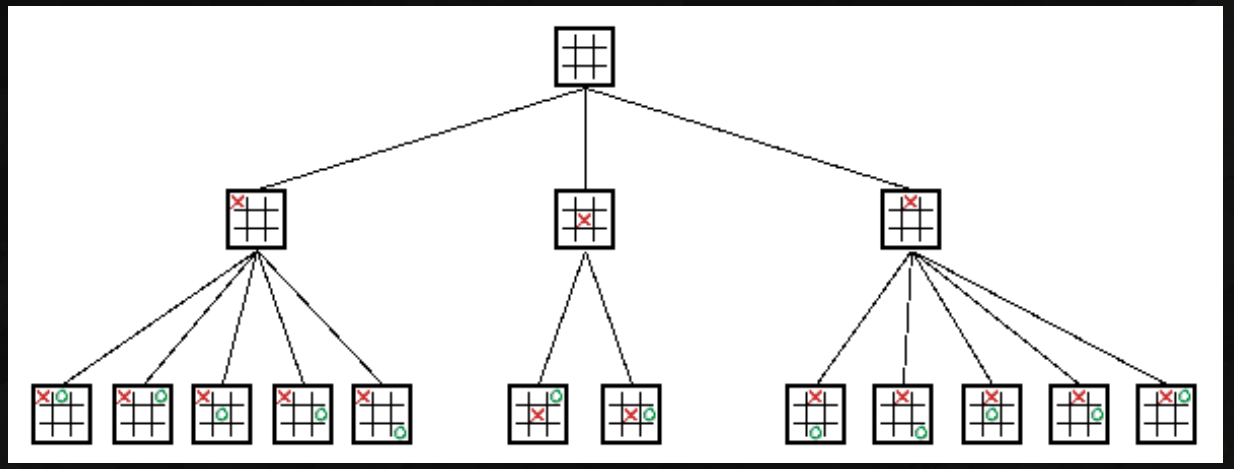
search algorithms

- **search**
  - look ahead and see what move to do next for optimal result
  - works well with known information (i.e., can see obstacles, pieces on board)
- **useful for turn-based strategy games and NPC pathfinding to target**

MinMax (Minimax) algorithm

- **popular tree search algorithm that applies to games where:**
  - players take turns
  - have perfect information (e.g., chess, checkers)
- **also works for games without perfect information or chance (Expectimax)**
  - add chance nodes
- **can work in real-time**

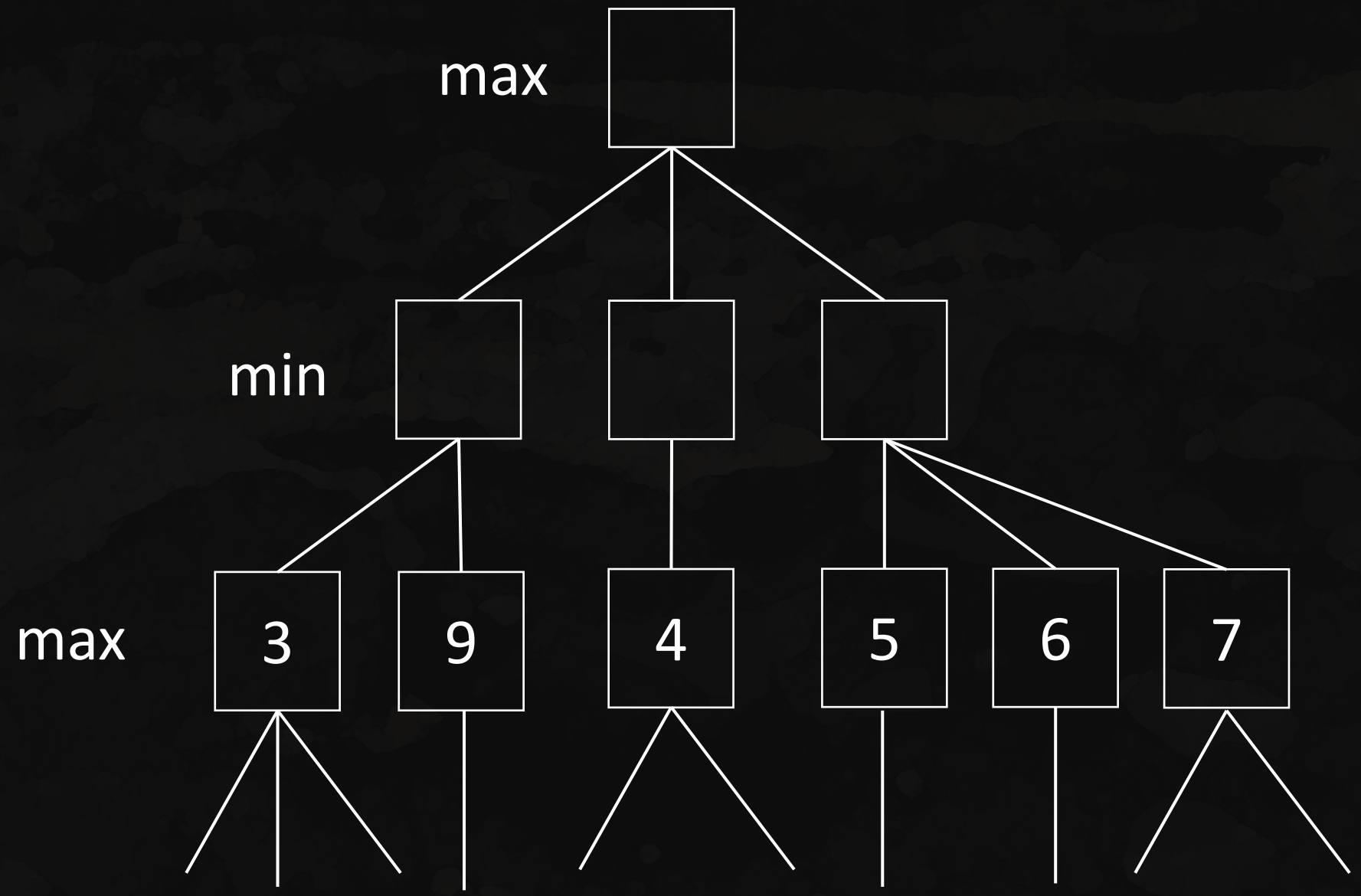
MinMax algorithm

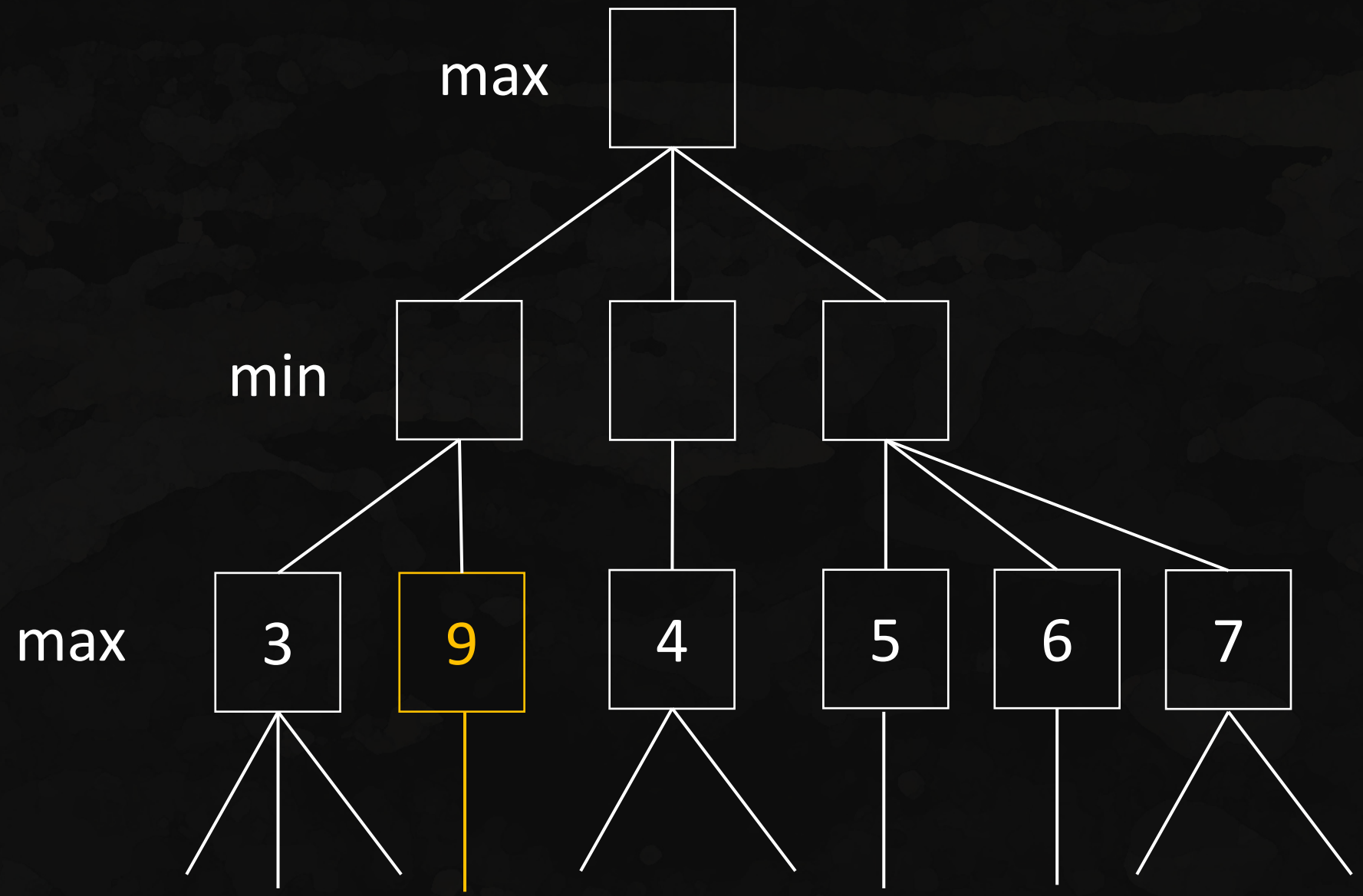


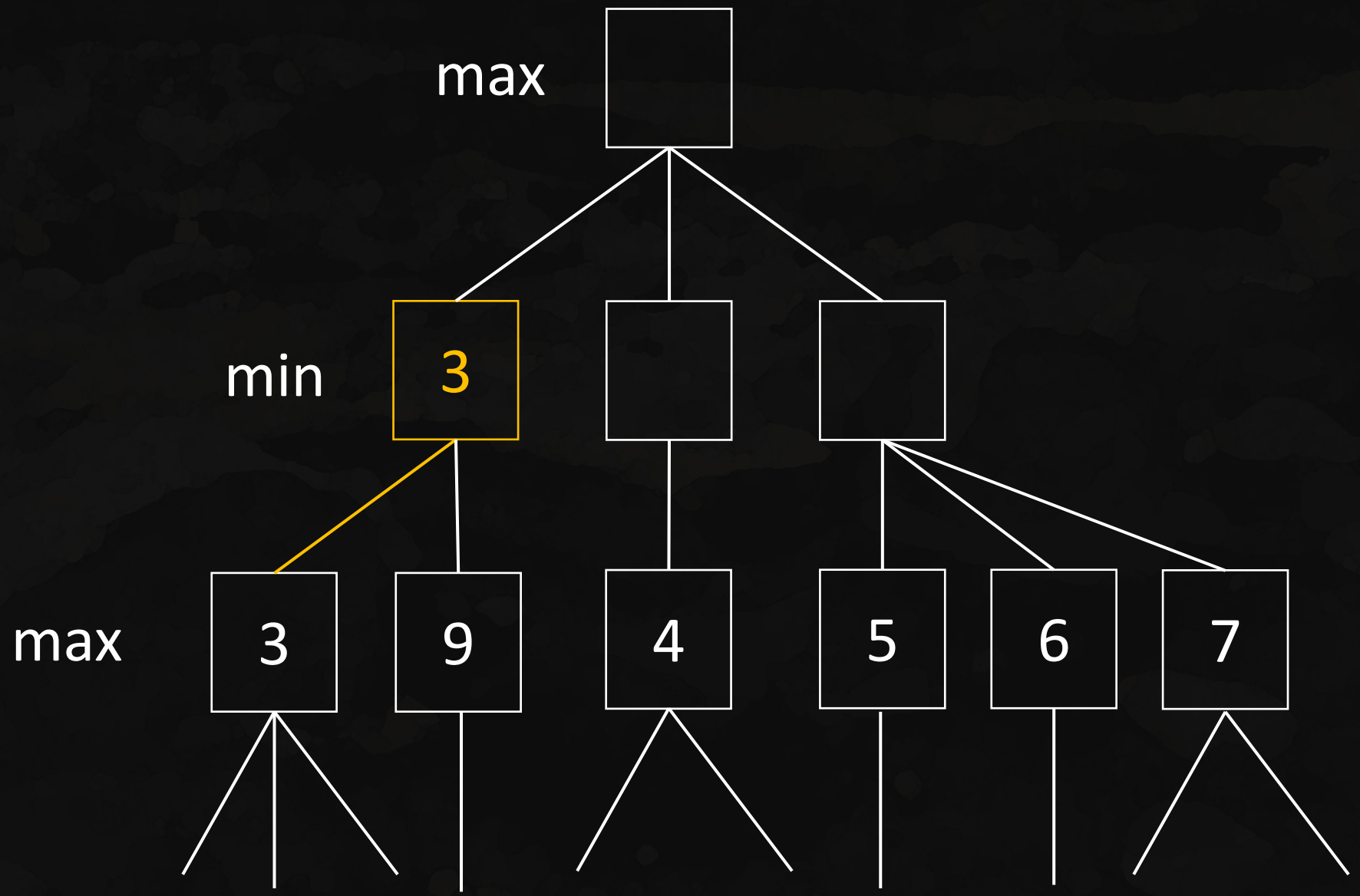
- **create an n-ary search tree of game states**
  - vertices represent decision states (i.e., after a move)
  - edges are decisions (i.e, the move)
- **assign value to each node: how good it would be for player to reach**
  - minimize the possible loss
- **alternate between max and min at each level**

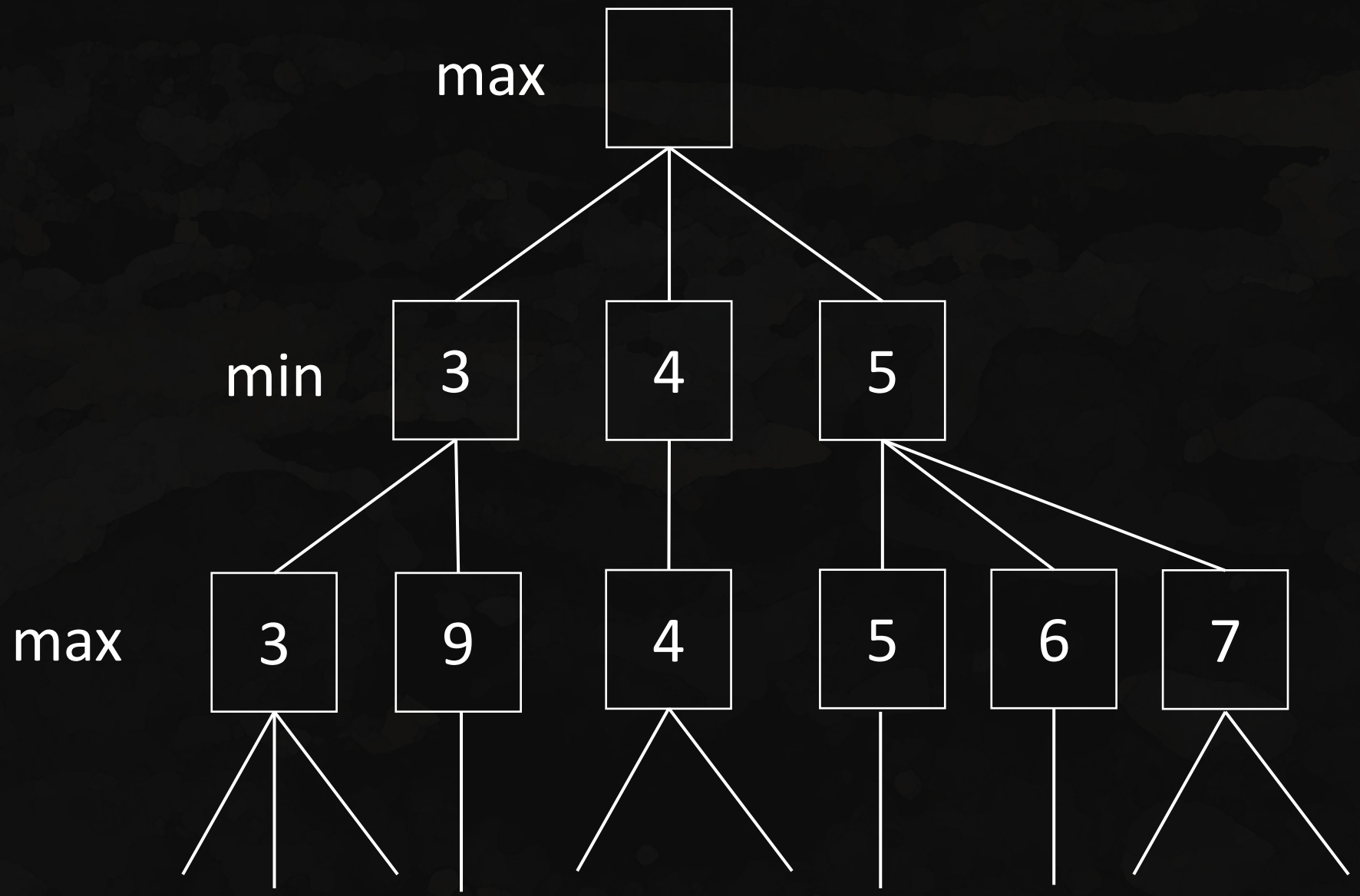
MinMax algorithm

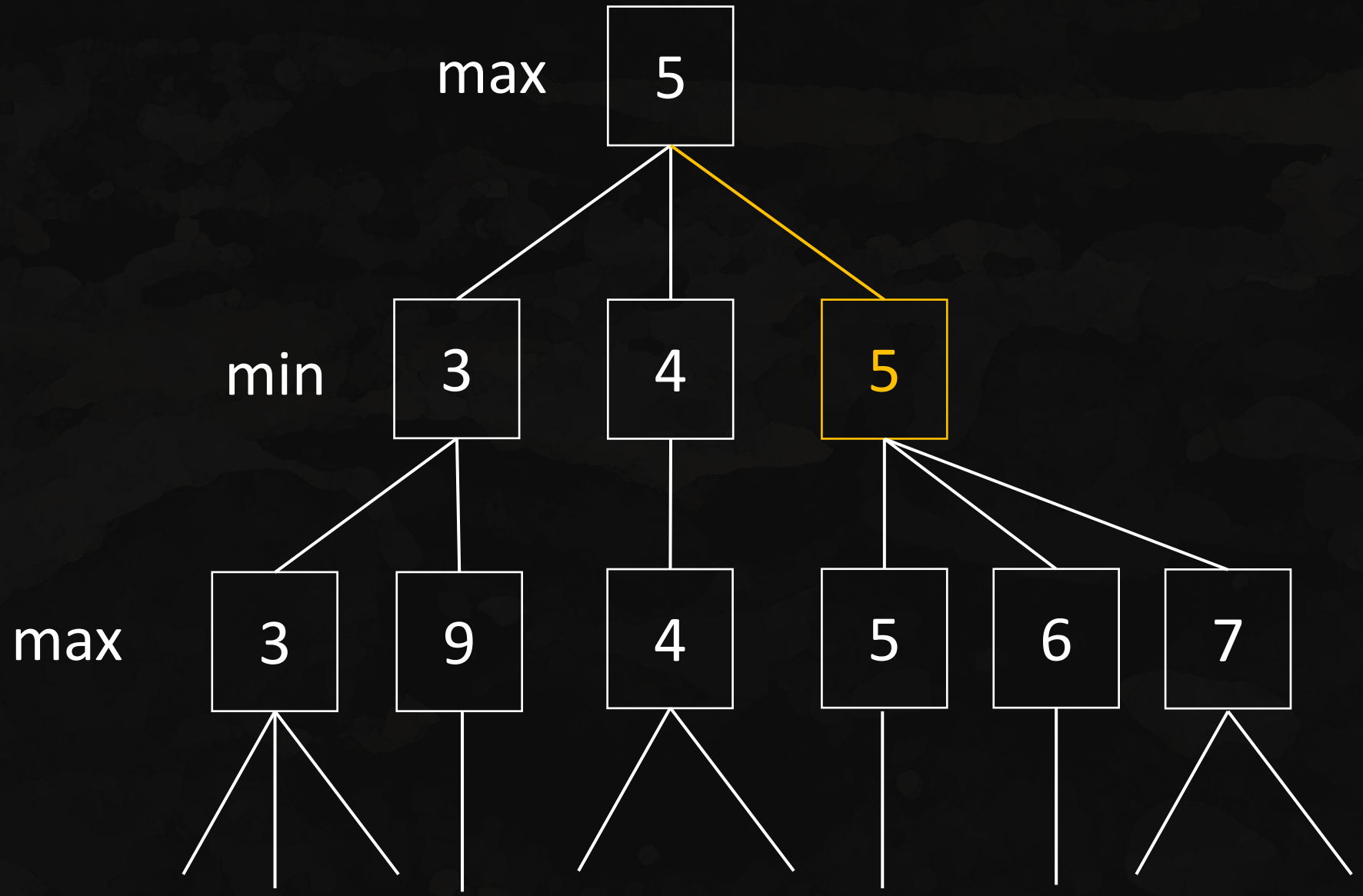
- **max (X) tries to maximize point value, while min (O) tries to minimize point value**
- **assume both players play to the best of their ability**
  - always make a move to minimize or maximize points
- **max will choose highest points, min will choose lowest**


max
min
max
3
9
4
5
6
7


max
min
max
3
9
4
5
6
7


max
min
3
max
3
9
4
5
6
7


max
min
3
4
5
max
3
9
4
5
6
7


max
5
min
3
4
5
max
3
9
4
5
6
7

MinMax and chess

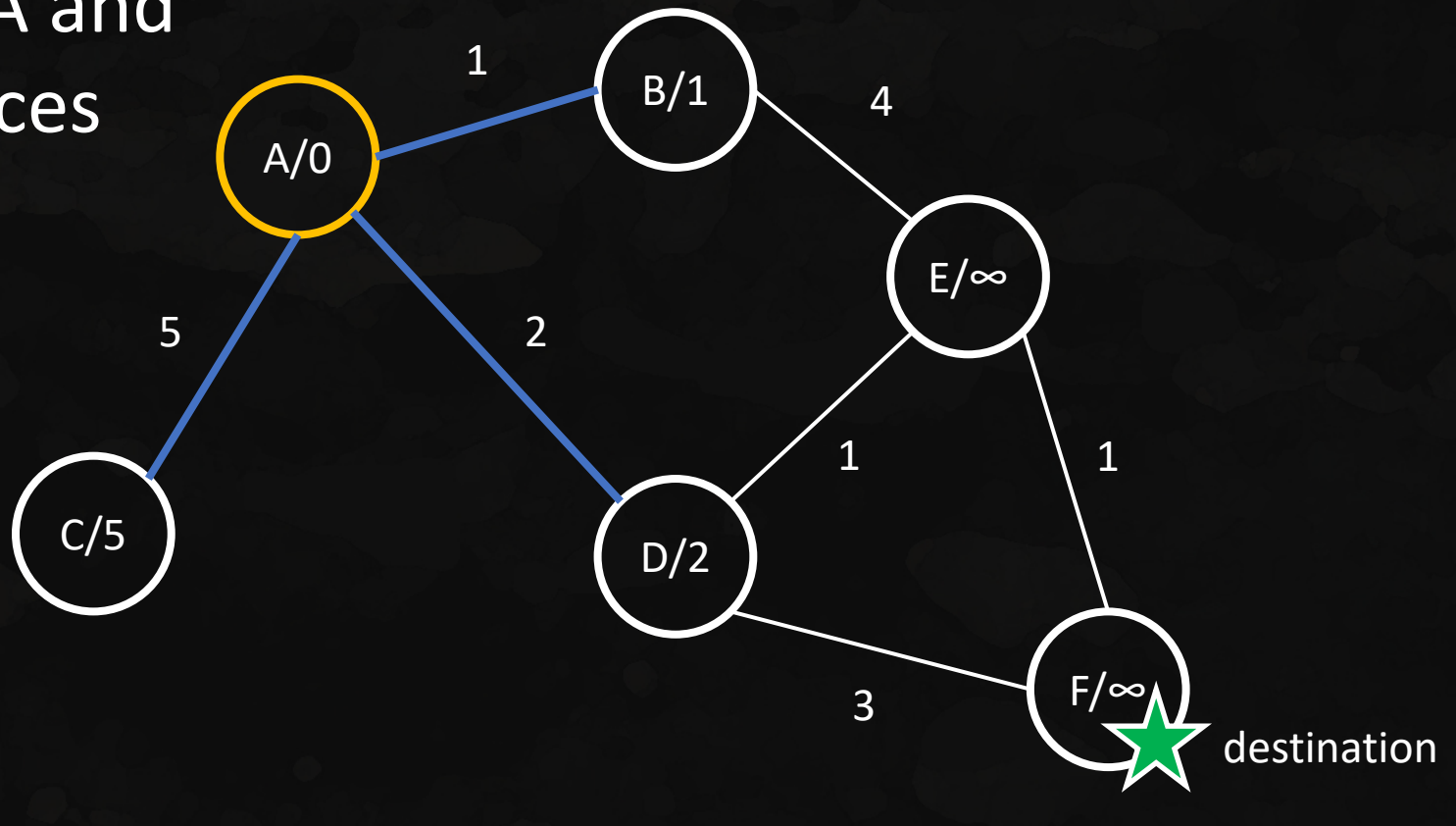
- **with full tree, one can determine best possible move**
- **however, full tree impossible for some games! (e.g., chess)**
  - at a given time, chess has ~35 legal moves.
  - exponential growth: $35^2 = 1225$ at two plies, $35^6 = 2$ billion, $35^{10} = 2$ quadrillion
  - games can last 40 moves or more, so 35^{40}
- **can't see end of game: must estimate**
  - evaluation function to guess end given current board

pathfinding

- **problem**
  - NPCs must figure out how to navigate from point A to point B in a 3D terrain in real time.
  - what is the most efficient way?
- **solution:**
  - *Dijkstra's algorithm*: determines shortest path from a vertex to all other vertices in a (weighted) graph

Dijkstra's algorithm: example

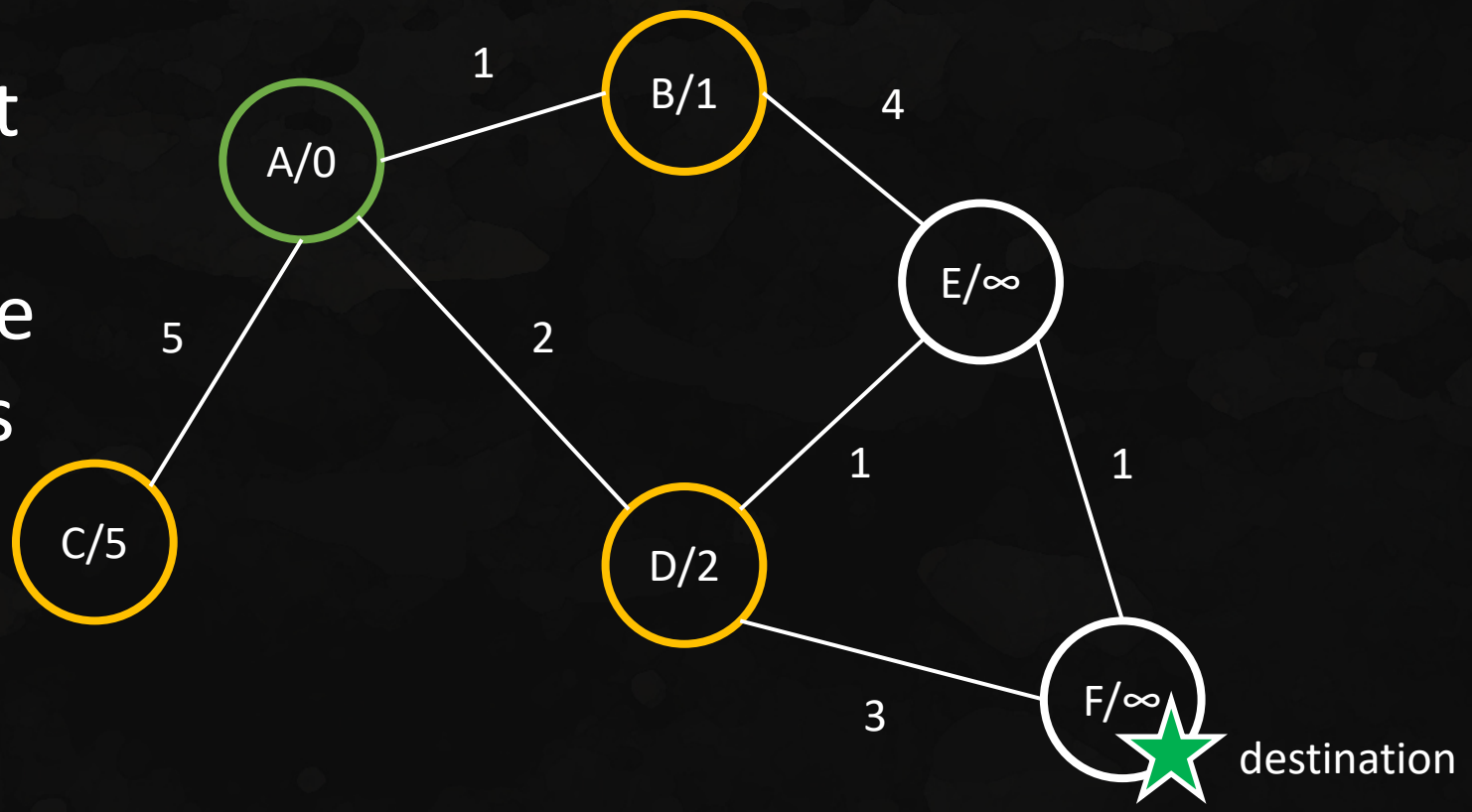
look at neighbors of A and calculate their distances



node under consideration

Dijkstra's algorithm: example

of the nodes under consideration, look at node with smallest distance and calculate distance to neighbors

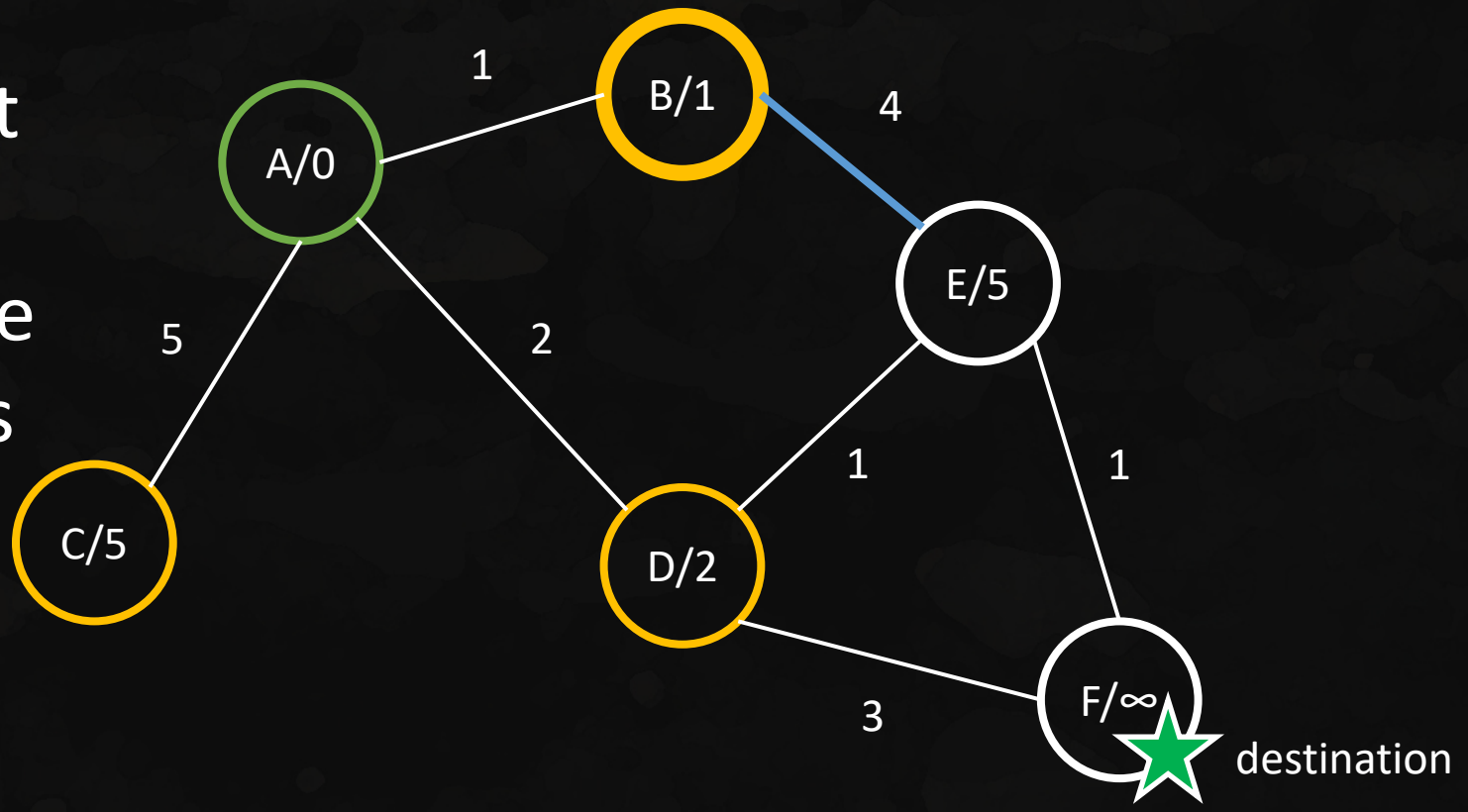


node under consideration

nodes visited

Dijkstra's algorithm: example

of the nodes under consideration, look at node with smallest distance and calculate distance to neighbors

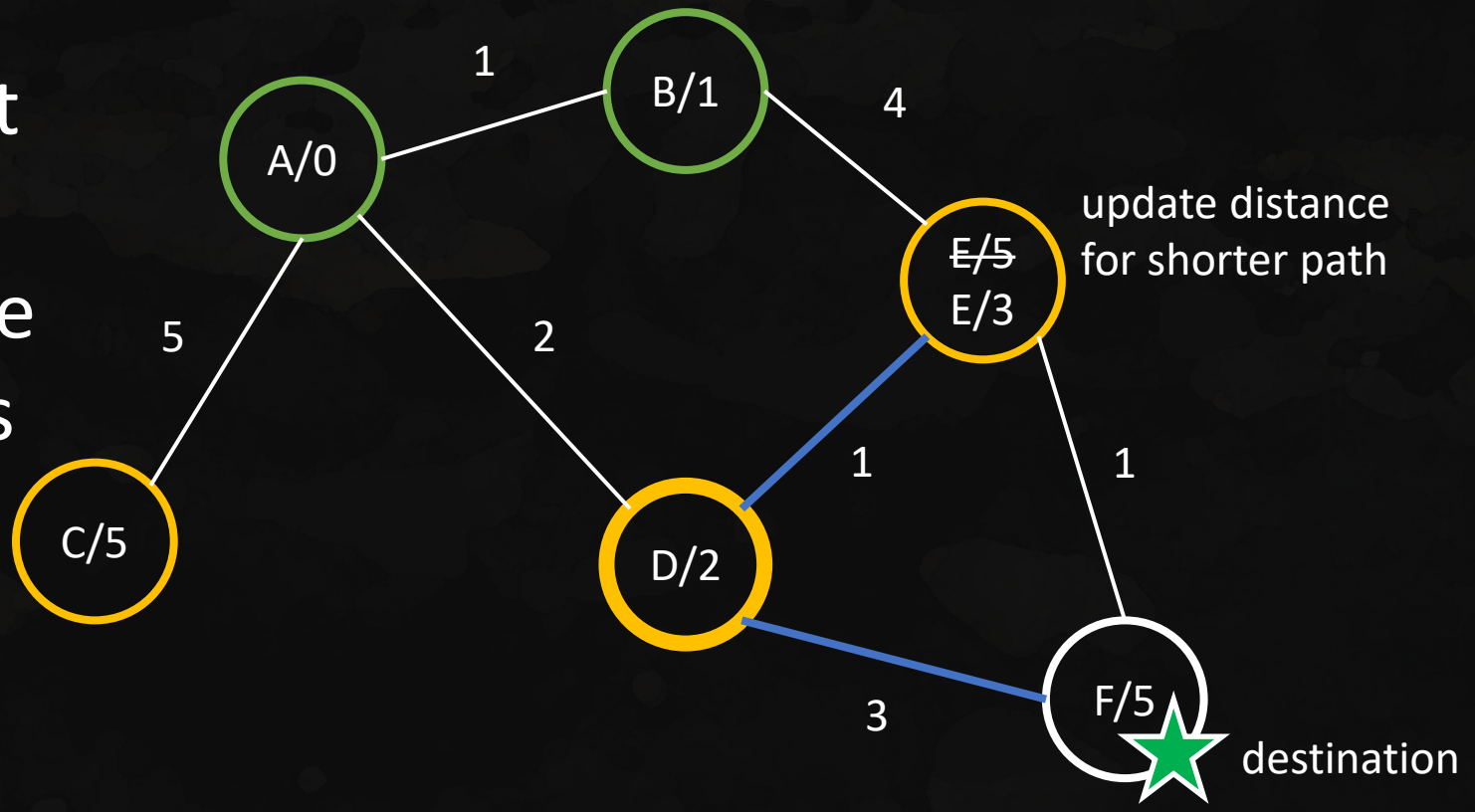


node under consideration

nodes visited

Dijkstra's algorithm: example

of the nodes under consideration, look at node with smallest distance and calculate distance to neighbors

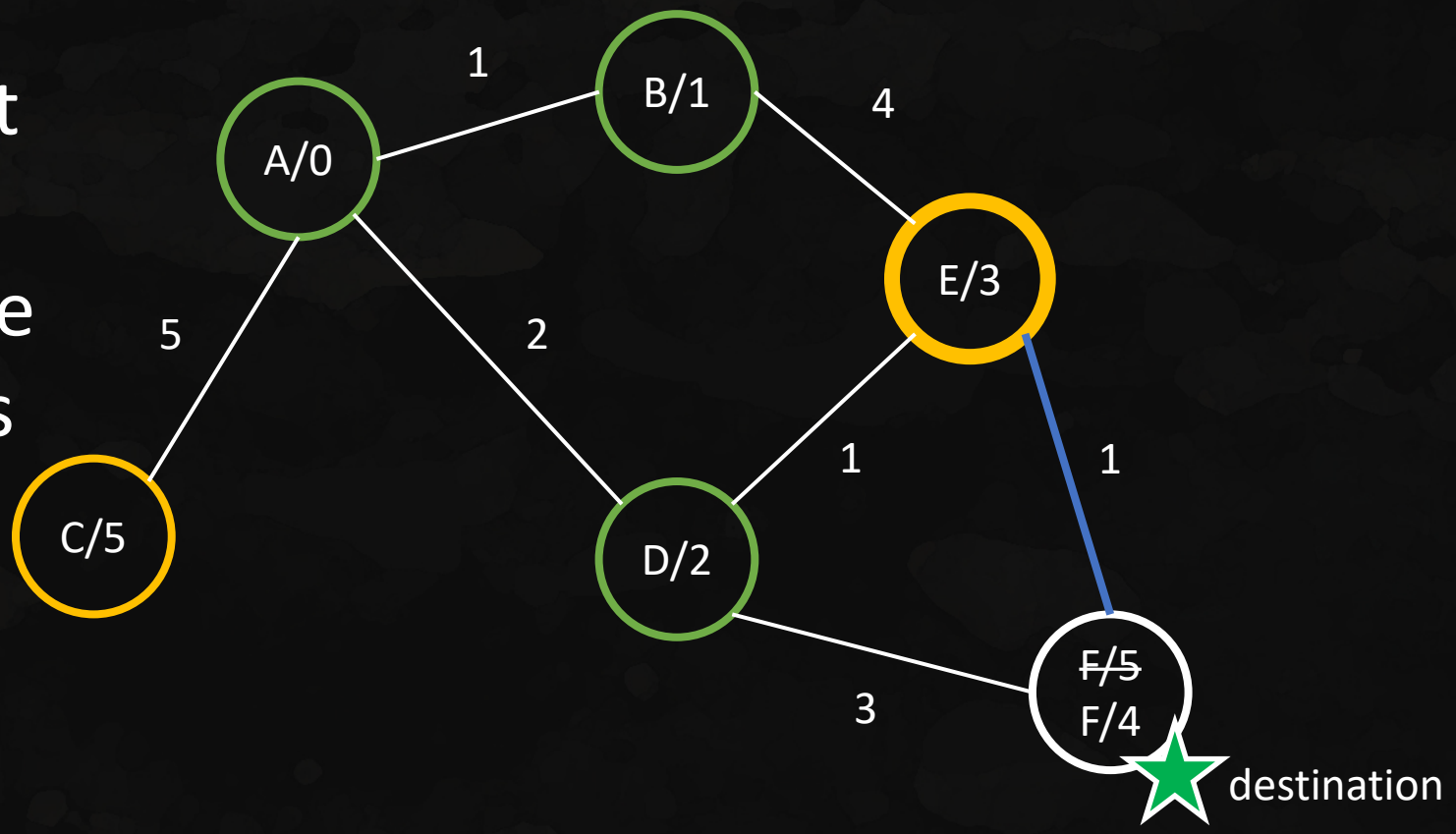


node under consideration

nodes visited

Dijkstra's algorithm: example

of the nodes under consideration, look at node with smallest distance and calculate distance to neighbors

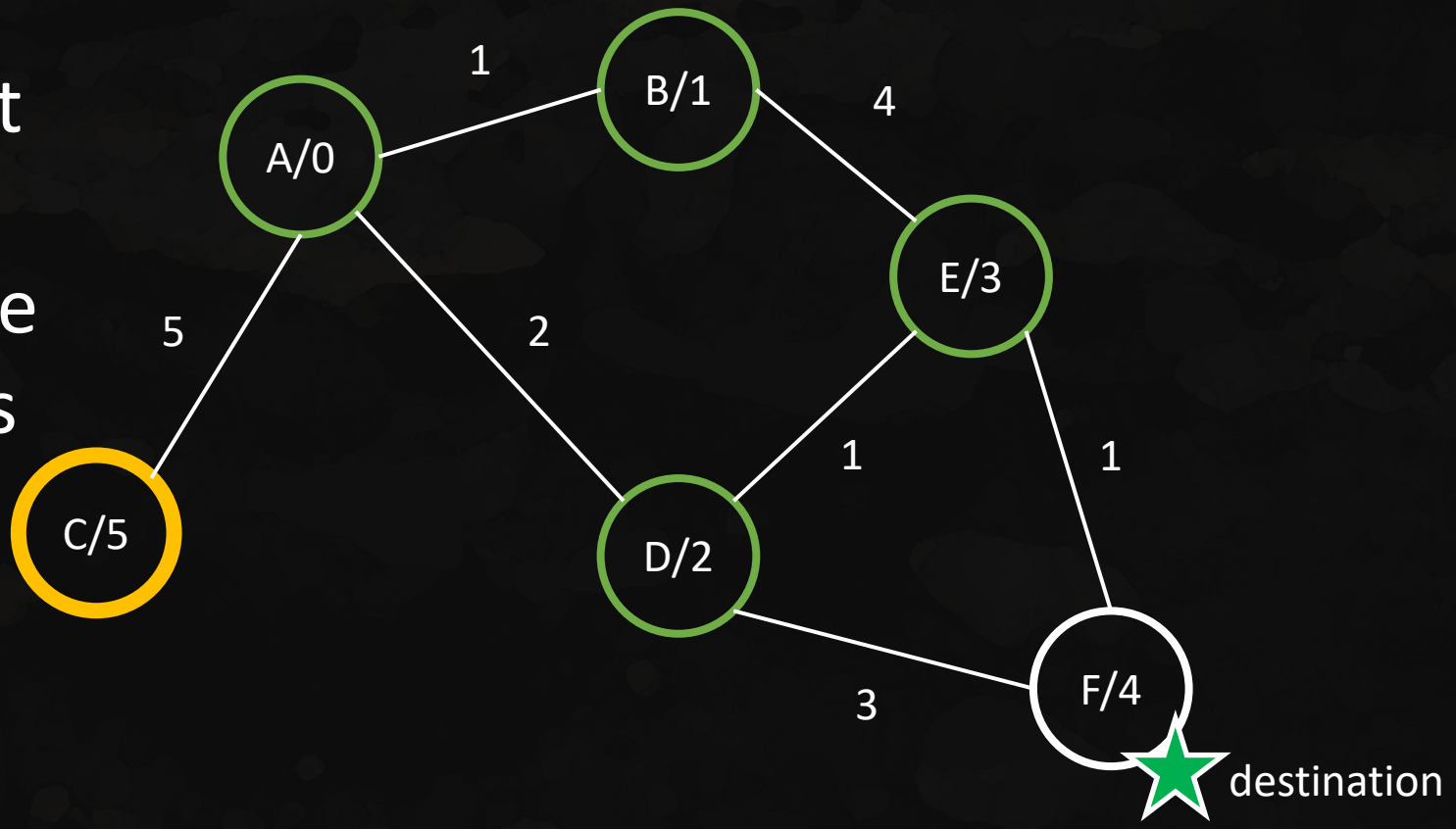


node under consideration

nodes visited

Dijkstra's algorithm: example

of the nodes under consideration, look at node with smallest distance and calculate distance to neighbors



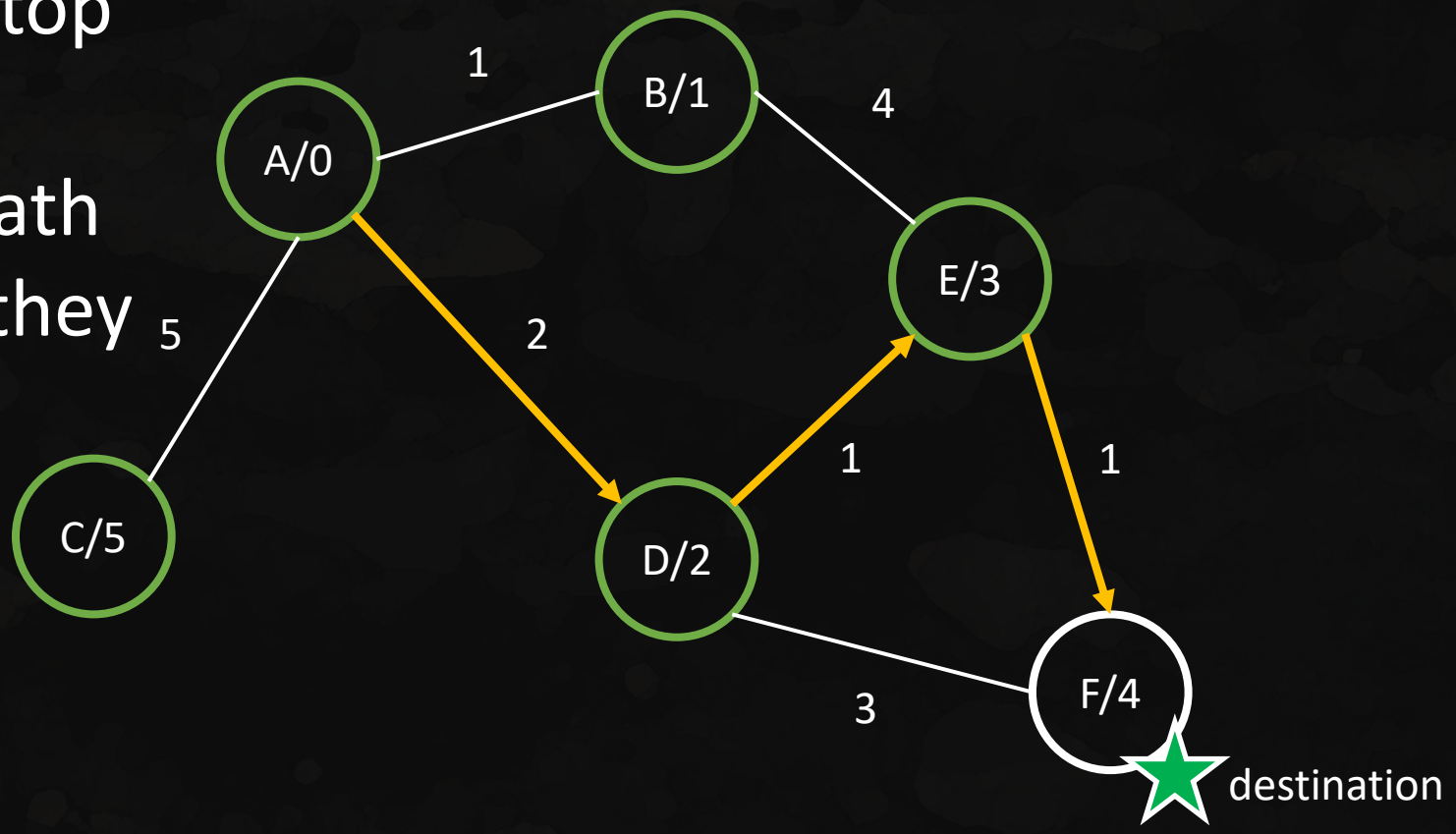
node under consideration

nodes visited

Dijkstra's algorithm: example

visited all nodes, so stop

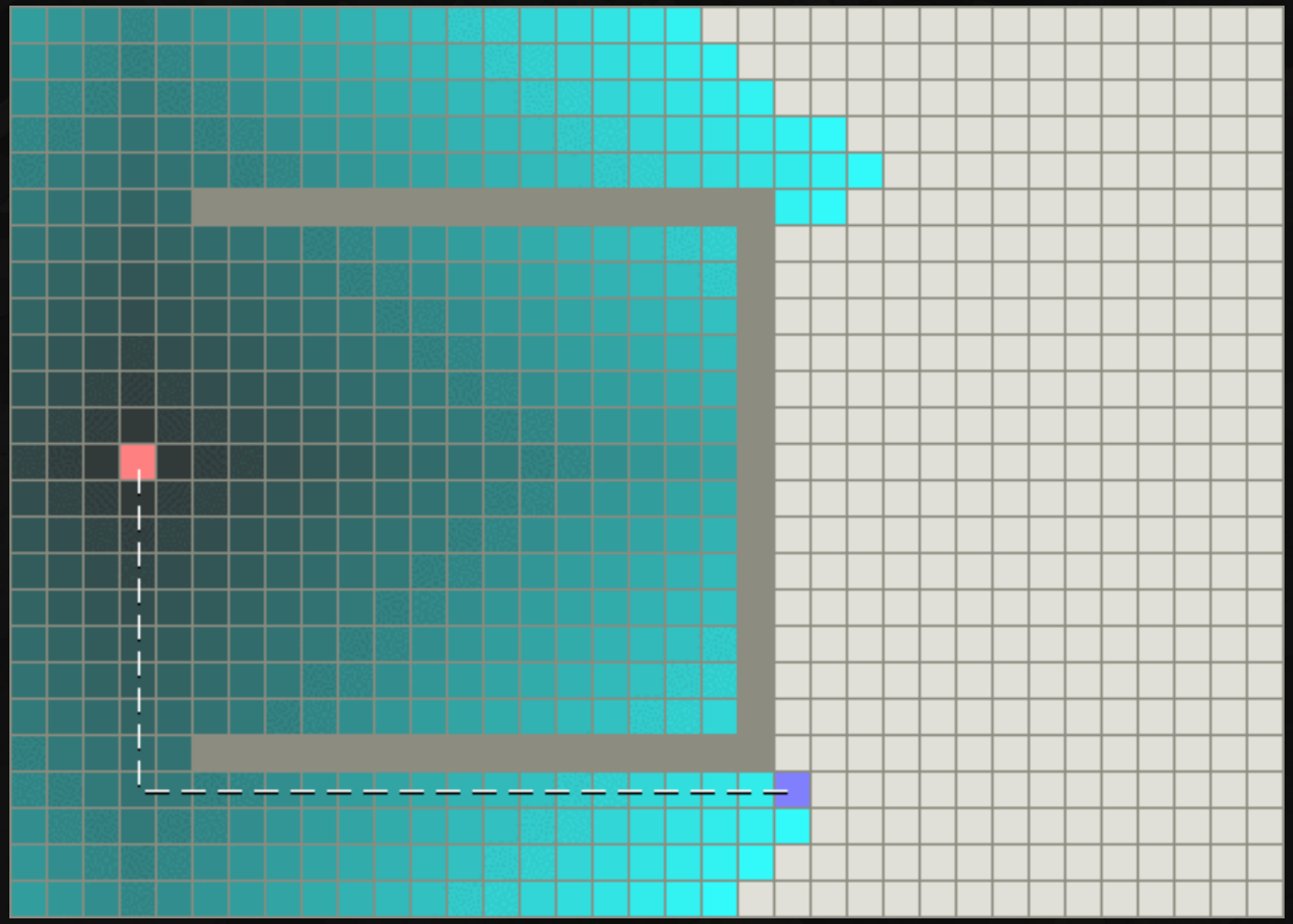
remember to store path info in each node as they are visited.



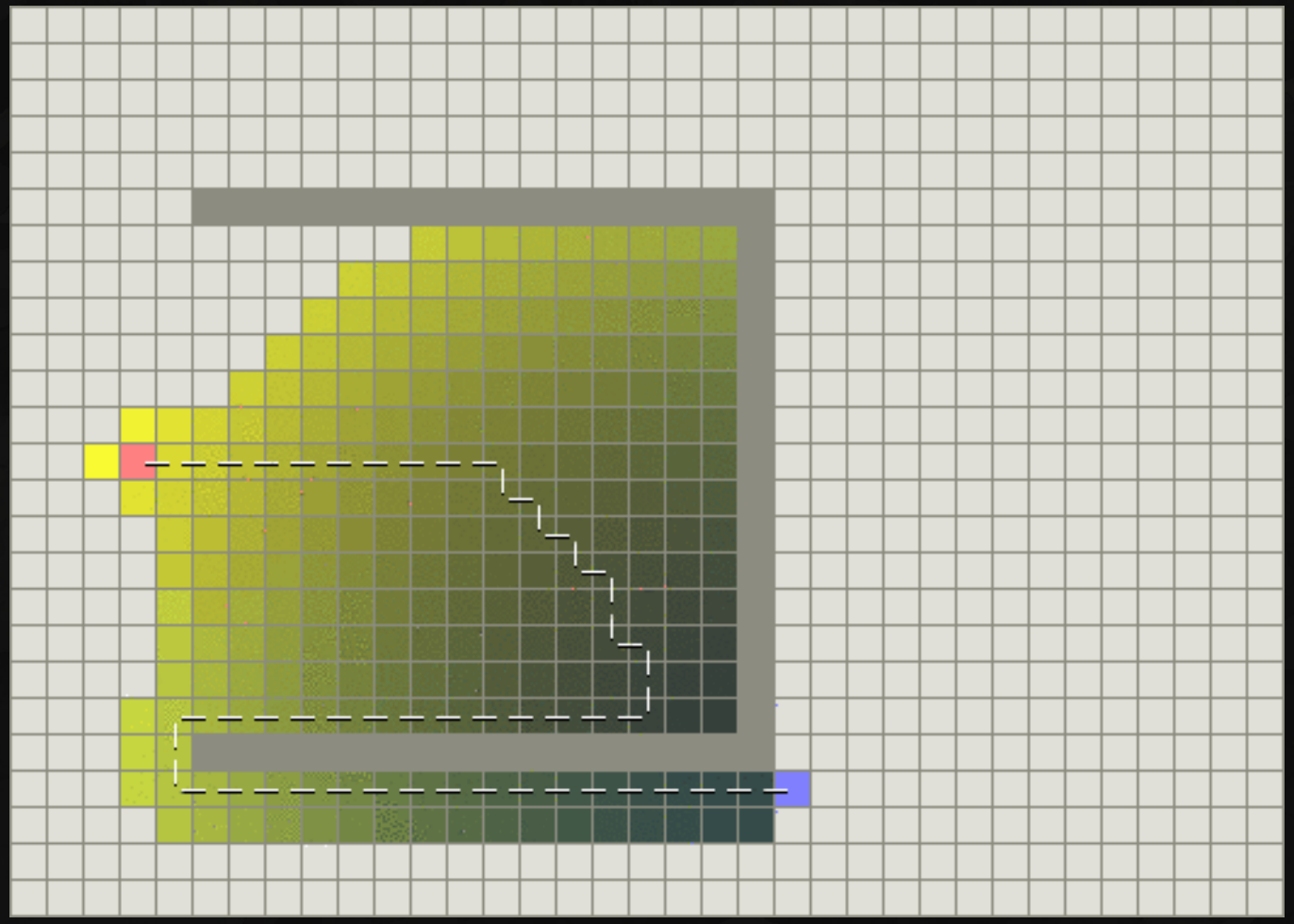
node under consideration

nodes visited

obstacles: Dijkstra's

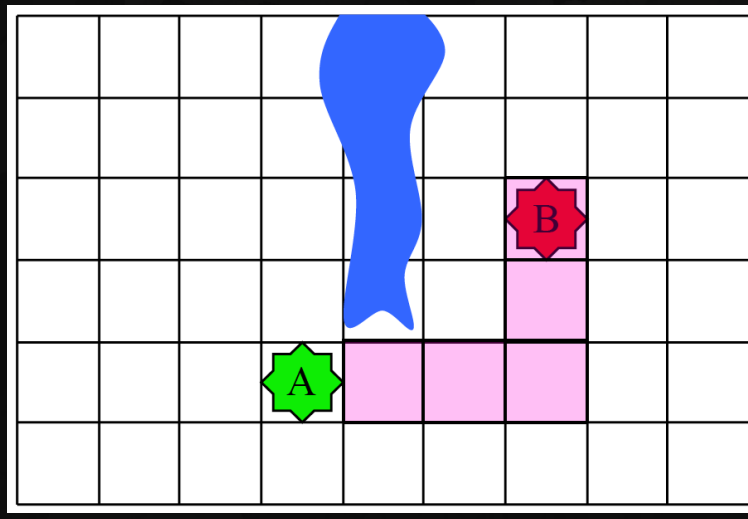


obstacles: Greedy Best-First-Search



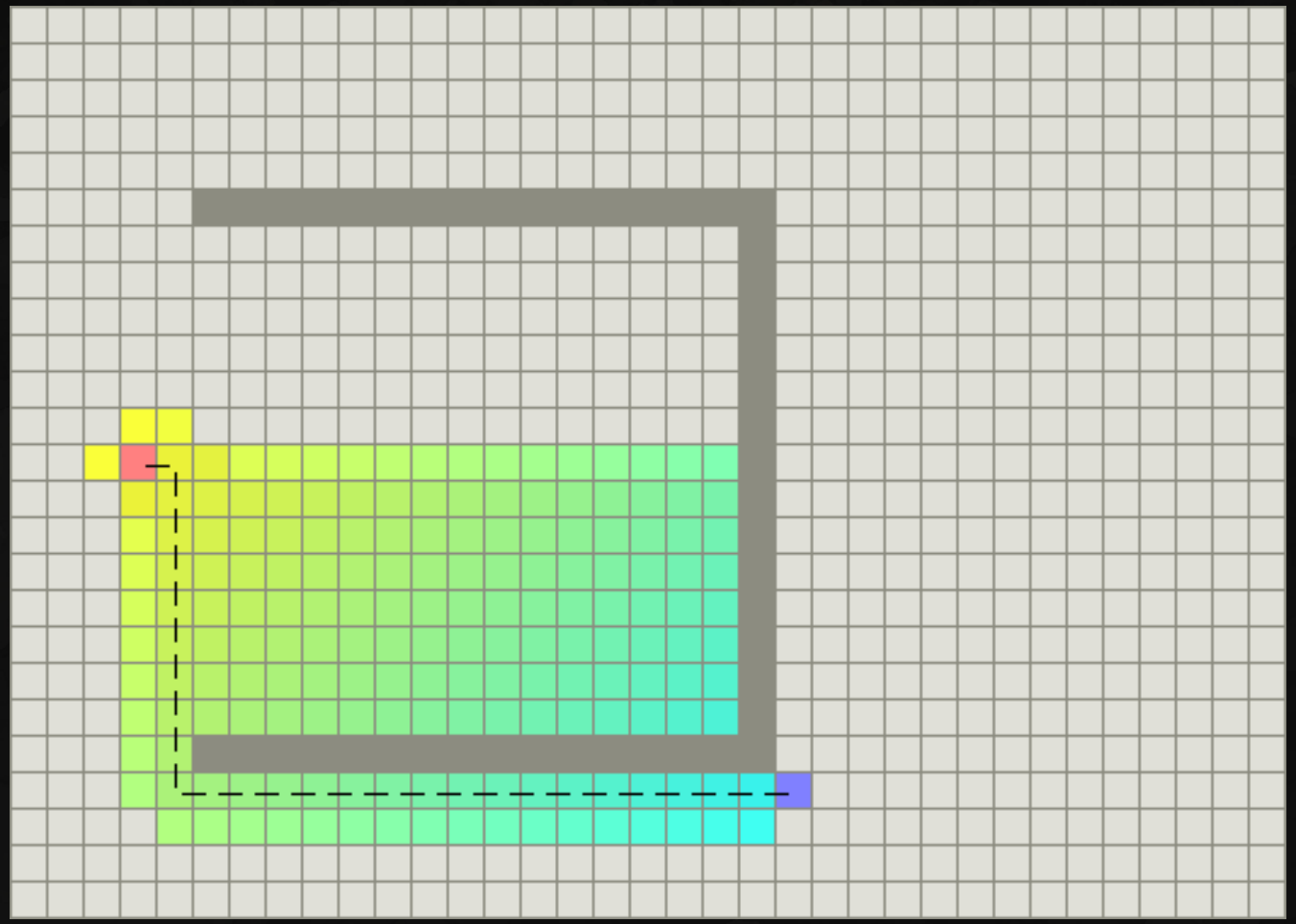
A* algorithm

- **A* is an informed search algorithm, enhancement of Dijkstra's**
  - the vertex to examine is determined by a heuristic of how close vertex is to the destination
- **f(n) = g(n) + h(n)**
  - minimize f(n)
  - g(n): cost to go from start vertex to n.
  - h(n): estimated cost of cheapest path from n to goal.
  - heuristic can be the "Manhattan distance"

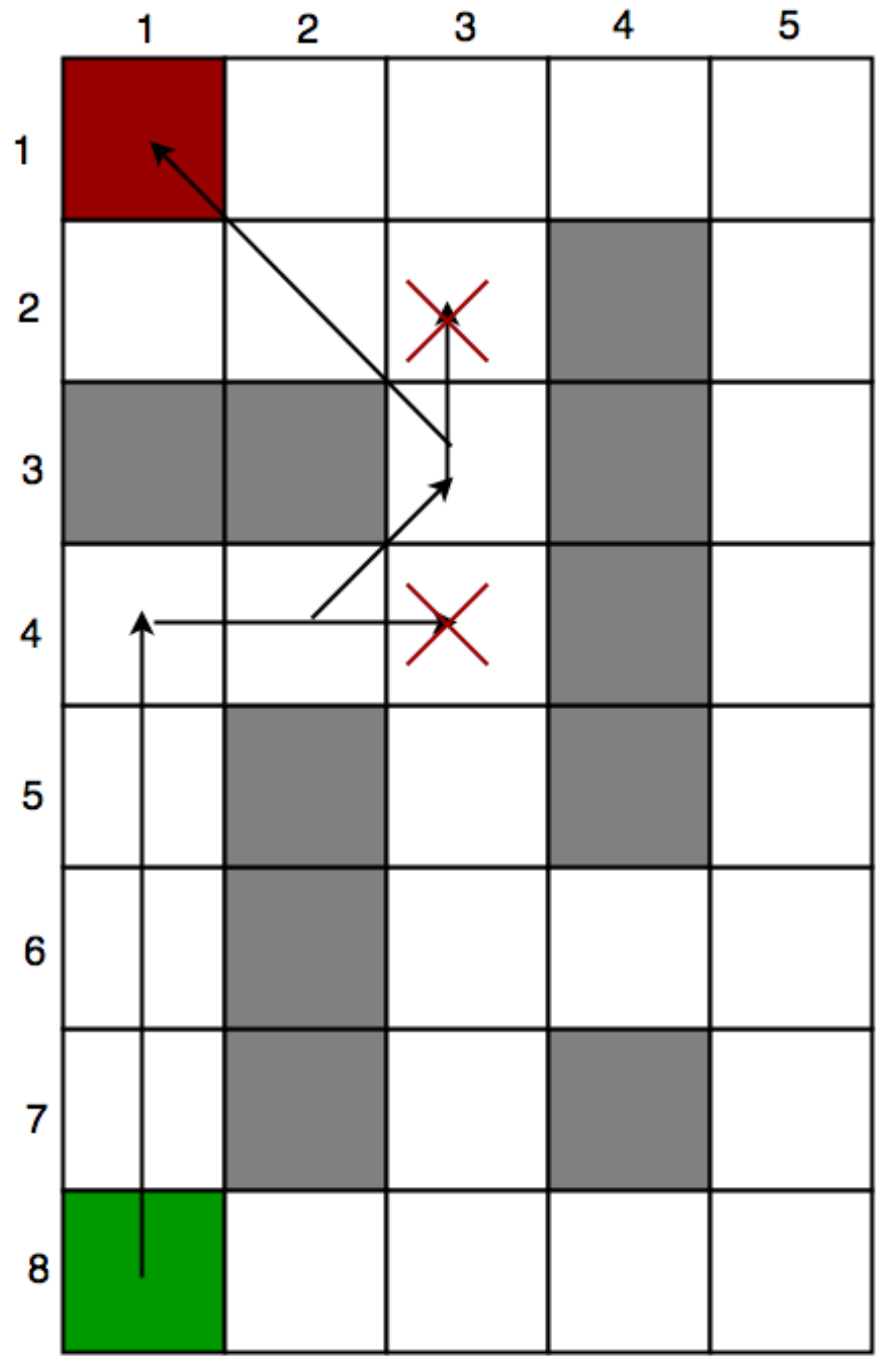


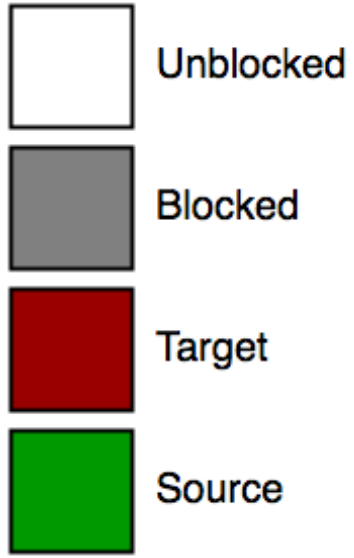
Manhattan distance = 50
30 (horizontal) + 20 (vertical)
10 per grid cell

obstacles: A*



A* algorithm





A* Search Algorithm makes the most intelligent choice at each step. Hence you can see that algorithm goes from (4,2) to (3,3) and not (4,3) (shown by cross).

Similarly the algorithm goes from (3,3) to (2,2) and not (2,3) (shown by cross).

pathfinding in complex terrain

- **how do you encode vertices for complex terrain?**
- **common techniques**
  - waypoint graphs
  - navigation meshes

waypoint graphs

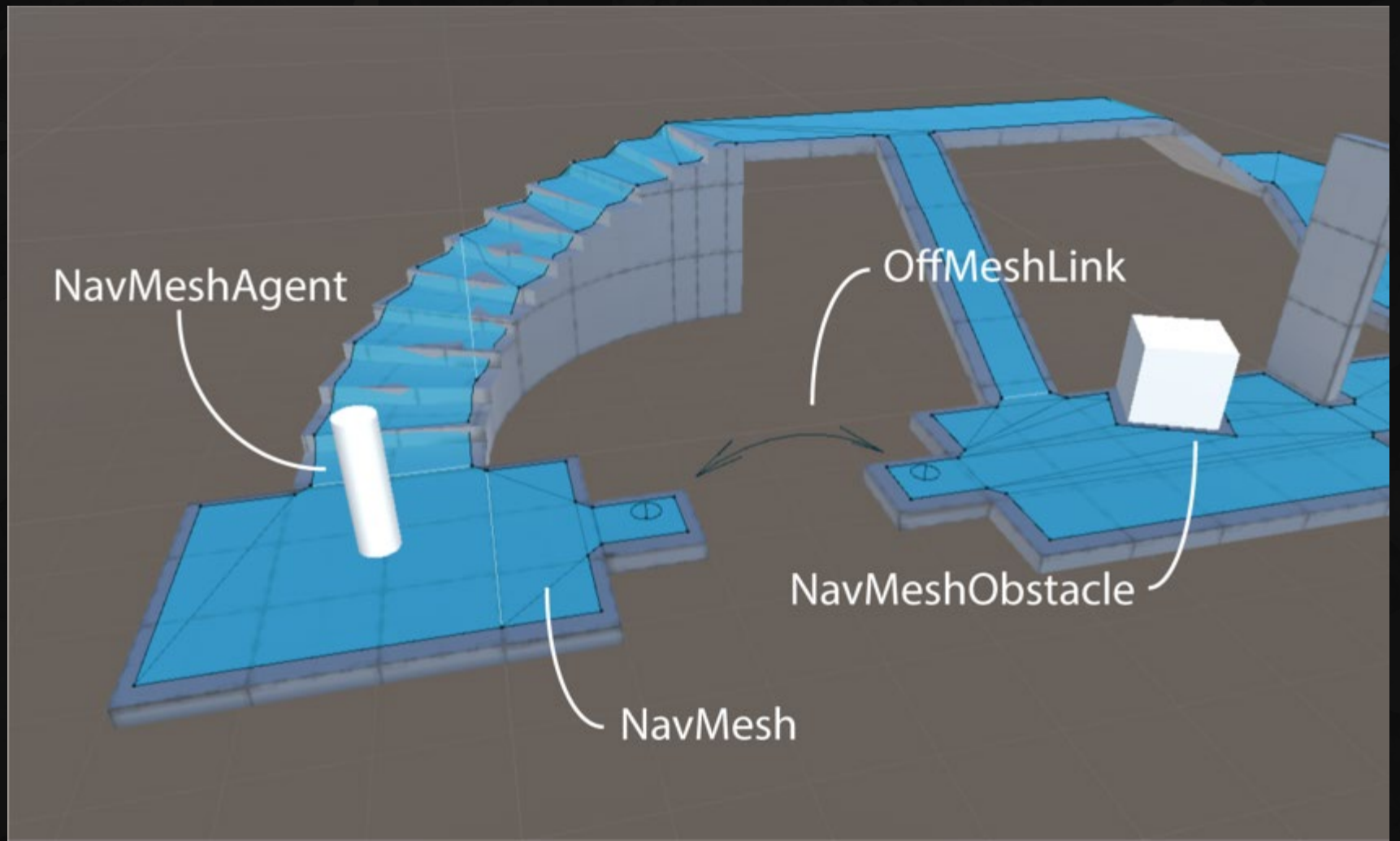
- specifies lines where an agent can safely traverse
- easy to create graph
- agent still needs to do collision detection with environment
  - costly for detailed environment
- for larger spaces, may need many vertices to avoid roundabout paths
- vertices can be used with A* rather than a massive grid

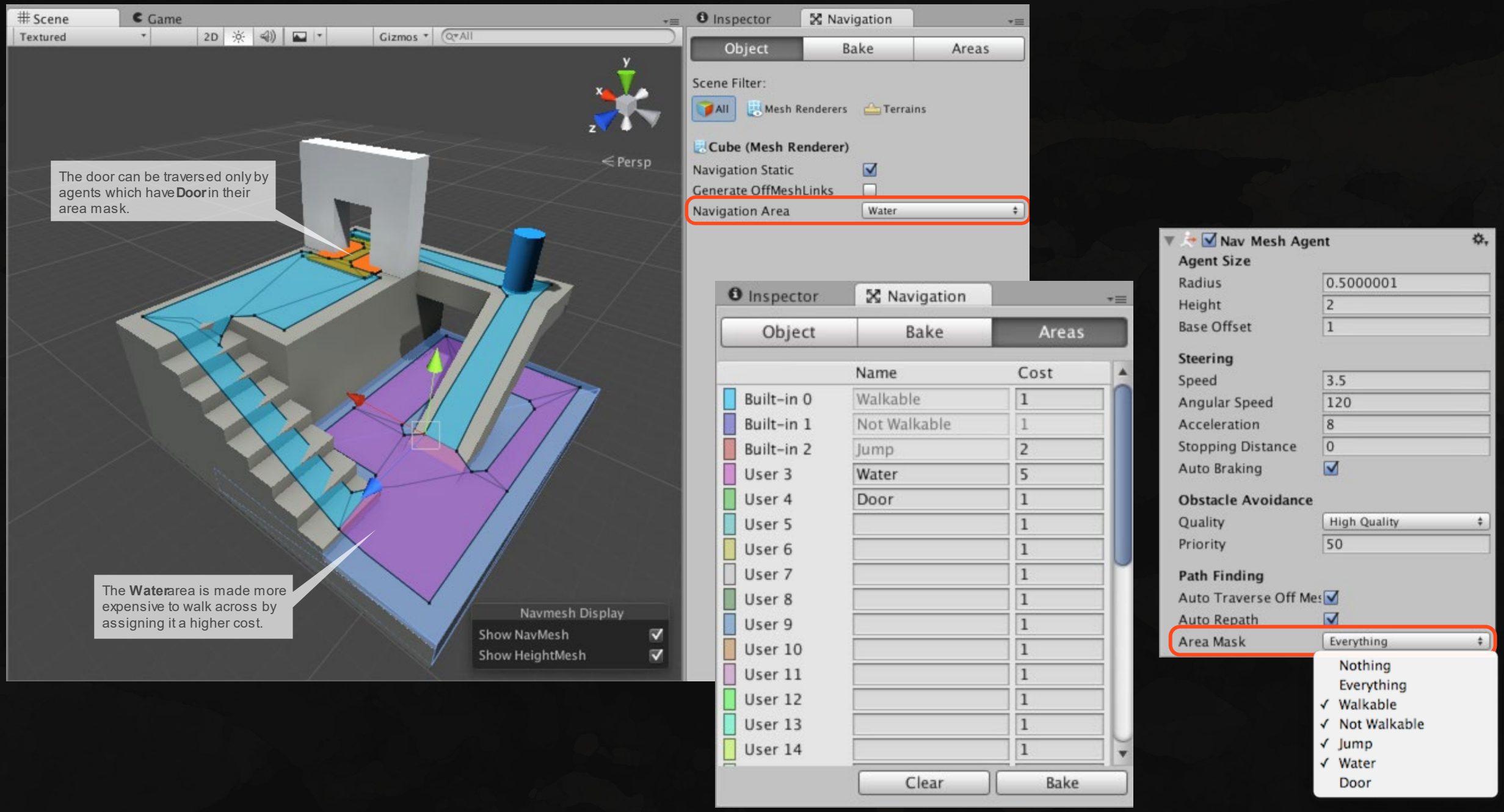
navigation meshes

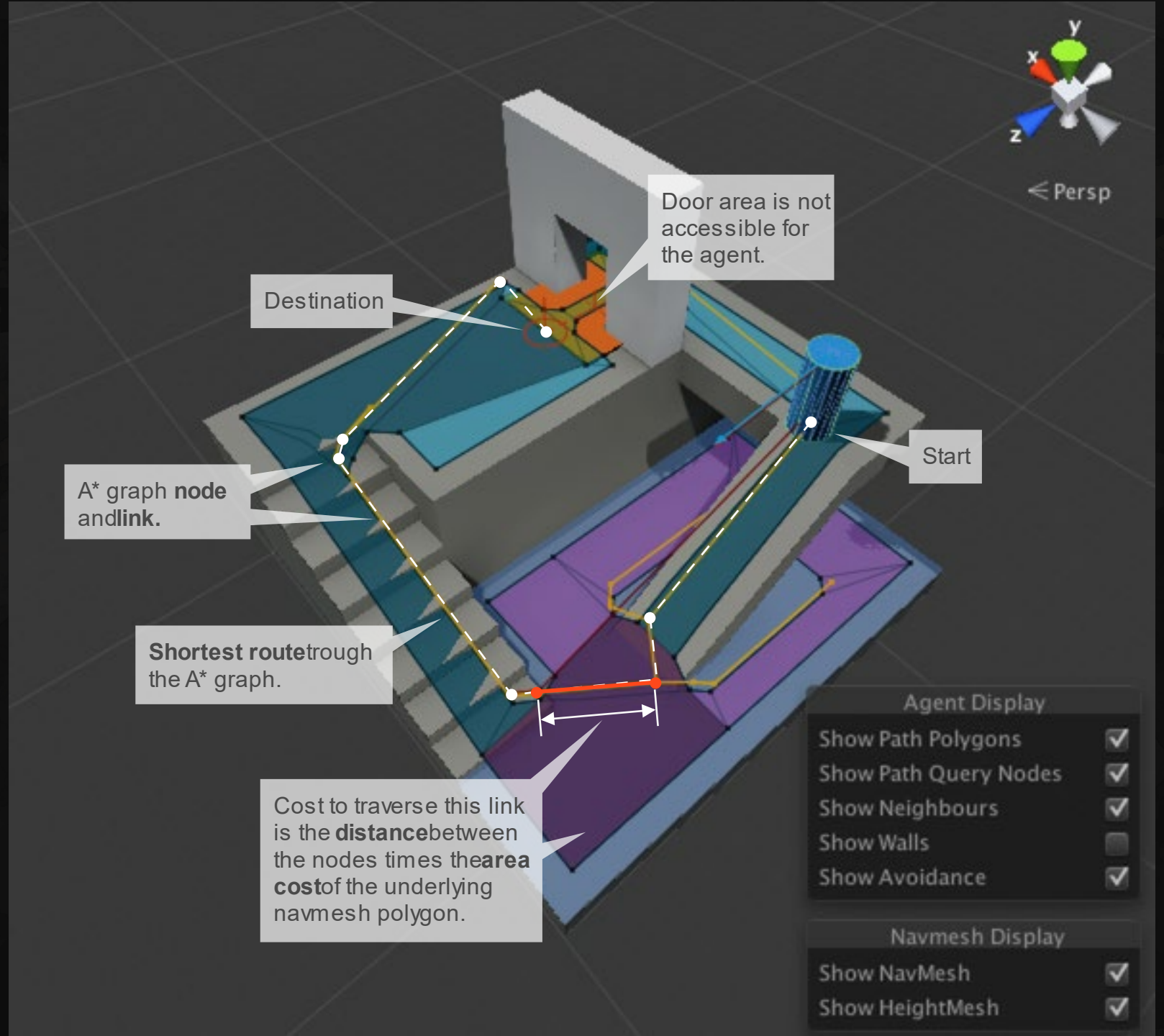
- **convex polygons represent walkable areas**
- **an edge is either:**
  - shared by another polygon (walkable)
  - not shared (unwalkable)
- **reduces collision detection to just the navigation mesh**
- **more work to construct mesh**

Unity NavMesh System

https://docs.unity3d.com/Manual/nav-NavigationSystem.html




Scene
Game
Textured
2D
Gizmos
Persp
The door can be traversed only by agents which have Door in their area mask.
The Water area is made more expensive to walk across by assigning it a higher cost.
Navmesh Display
Show NavMesh
Show HeightMesh
Inspector
Navigation
Object
Bake
Areas
Scene Filter:
All
Mesh Renderers
Terrains
Cube (Mesh Renderer)
Navigation Static
Generate OffMeshLinks
Navigation Area
Water
Name
Cost
Built-in 0
Walkable
Built-in 1
Not Walkable
Built-in 2
Jump
User 3
Water
User 4
Door
User 5
User 6
User 7
User 8
User 9
User 10
User 11
User 12
User 13
User 14
Clear
Bake
Nav Mesh Agent
Agent Size
Radius
0.5000001
Height
2
Base Offset
1
Steering
Speed
3.5
Angular Speed
120
Acceleration
8
Stopping Distance
0
Auto Braking
Obstacle Avoidance
Quality
High Quality
Priority
50
Path Finding
Auto Traverse Off Mes
Auto Repath
Area Mask
Everything
Nothing
Everything
Walkable
Not Walkable
Jump
Water
Door


y
x
z
Persp
Door area is not accessible for the agent.
Destination
Start
A* graph **node** and **link.**
Shortest route trough the A* graph.
Cost to traverse this link is the **distance** between the nodes times the **area cost** of the underlying navmesh polygon.
Agent Display
Show Path Polygons
Show Path Query Nodes
Show Neighbours
Show Walls
Show Avoidance
Navmesh Display
Show NavMesh
Show HeightMesh

lower-tech approaches

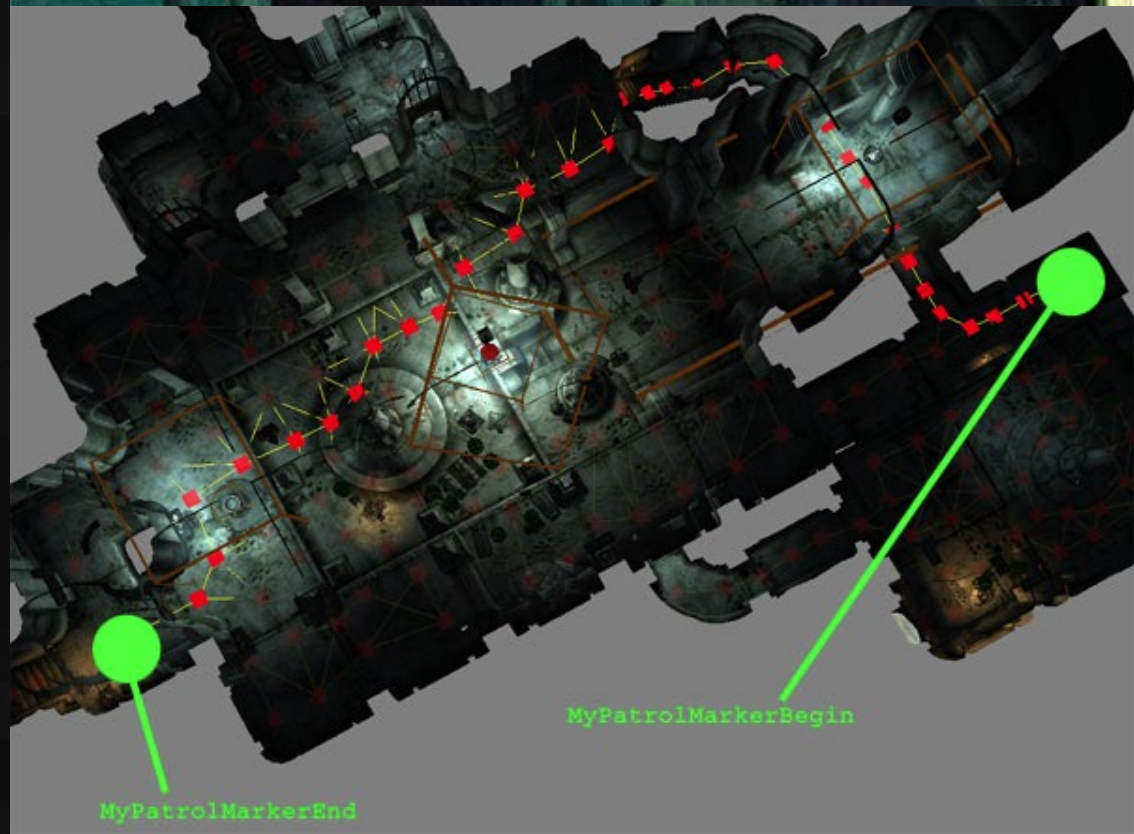
- previous approaches mentioned may be too costly
- several simpler solutions depending on AI needs
- mostly used in early games to save processing power and memory
- can be used in combination with previous approaches

digital scent

- digital scent that fades away with time
- when player character walks about the space, they leave a vector path that fades with time
- NPCs can determine if they are close enough to a vertex of the trail to follow
- good if the player character runs away and NPC needs pathfinding without using A*

patrolling AI

- **pre-defined patrol routes**
  - when agent is in idle mode, it follows pre-defined path
  - multiple patrol paths can reduce repetitiveness of agent
  - common in stealth games
  - can use waypoint graph



patrolling AI: example

- **if idle, perform 1 of N pre-defined patrol routes.**
- **if player beyond a certain distance from AI turn off AI's proximity sensors.**
- **with proximity sensors:**
  - if detect player/enemy, jump to "attack" state and move towards
  - if attacked, jump to "evade" state and move perpendicular
  - if player beyond certain range, follow player's scent
  - if agent loses scent but still need pursuit, cheat by computing A*
  - else resume patrol routes: how to get back to old patrol routes?
    agent keeps own scent so it can follow it back to previous patrol route.

load balancing

- agents may need to keep compute load down.
- give less processing to far away agents or agents idling.

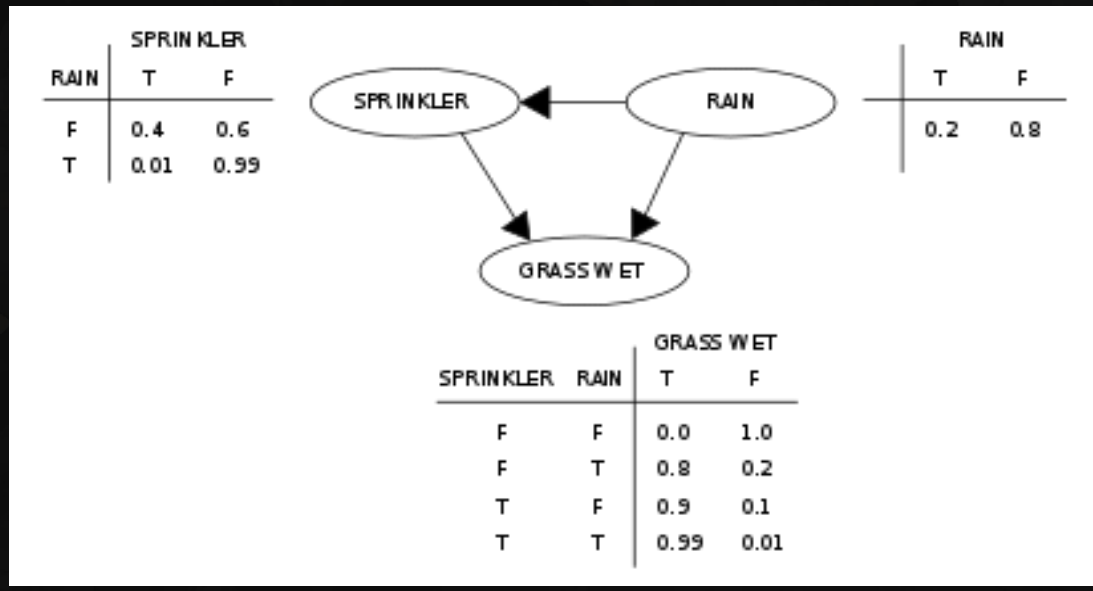
adaptive AI

- **learn from player behaviors**
  - good for combat,
    e.g. fighting games
- **approaches**
  - Bayesian networks
  - genetic algorithms
  - reinforcement learning

Bayesian networks

- **directed acyclic graphs**
  - vertices represent variables.
  - edges represent conditional dependencies.
  - no edges between vertices represent conditional independencies between the variables.

- **nodes can represent any kind of variable (e.g., a measured parameter or a hypothesis).**



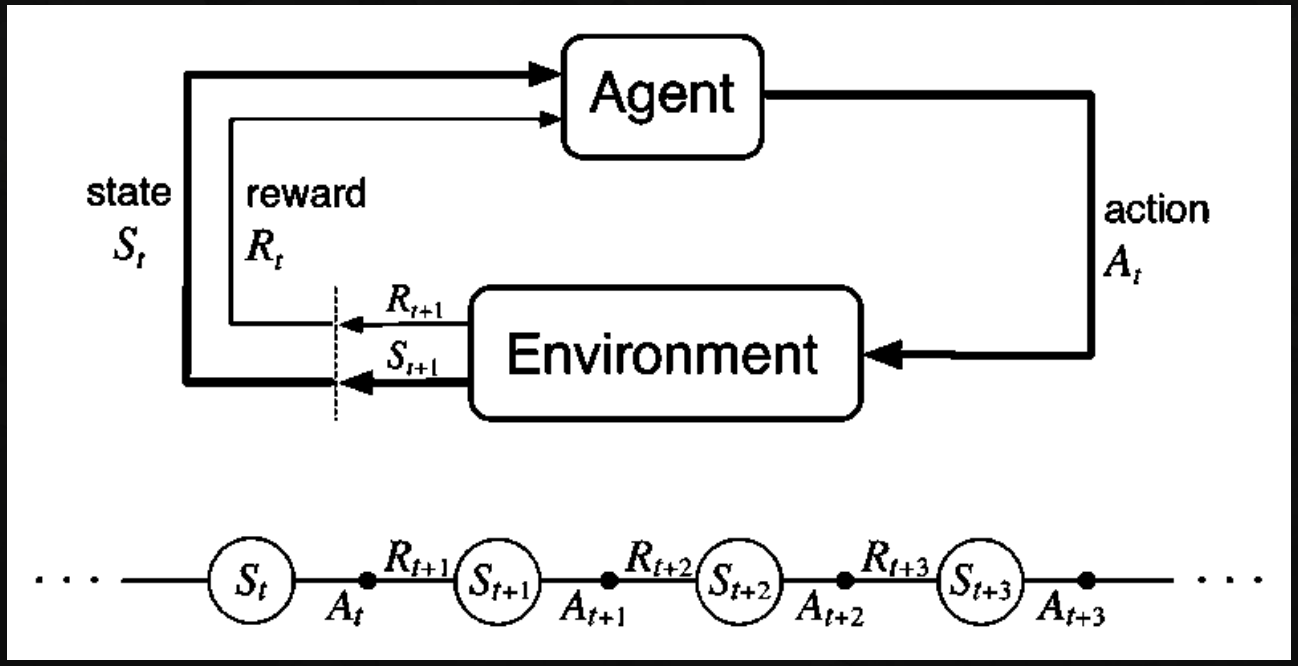
Bayesian network

- **in a fighting game, predict a player's third move given the first and second move**
  - three moves: punch, low kick, and high kick
  - build a table where the first column represents the first move and the second column represents the third move of the player
  - the third column of the table is split into 3 columns, one for each possible move, and each entry represents the count of the third move of the player given the first two moves.

1st move	2nd move	3rd move		
		punch	low kick	high kick
punch	punch	10	15	2
punch	low kick	1	9	12
punch	high kick	6	0	1
low kick	punch	8	0	1
low kick	low kick	2	3	1
low kick	high kick	23	1	13
high kick	punch	4	5	9
high kick	low kick	1	2	1
high kick	high kick	0	8	9

reinforcement learning

- problems involving agent interacting with environment with numeric reward signals
- goal: learn how to take actions in order to maximize reward



Atari games

[Playing Atari with Deep Reinforcement Learning]

- **objectives:** complete the game with the highest score
- **state:** raw pixel inputs of the game state
- **action:** game controls (e.g., left, right, up, down)
- **reward:** score increase/decrease at each time step

Google’s DeepMind playing Breakout

https://www.youtube.com/watch?v=V1eYniJ0Rnk

Unity ML-Agents Toolkit

- https://github.com/Unity-Technologies/ml-agents
- enable games and simulations to serve as environments for training intelligent agents
- reinforcement learning, imitation learning, neuroevolution

questions?