GameJam

Short Introduction on (Symbolic) AI Methods

Disclaimer

- This presentation focuses on the usage of symbolic AI techniques for level generation and action selection since they:
  - Often dont require that much computational power
  - No training and not that much parameter tuning
  - Allow good control over the desired outputs if the problems are well defined
- We don't talk about generative AI/Deep Learning techniques
  - Often way to compute intensive
  - Training not feasible in short projects
  - Hard to handle/control, especially if they need to fit into a larger concept

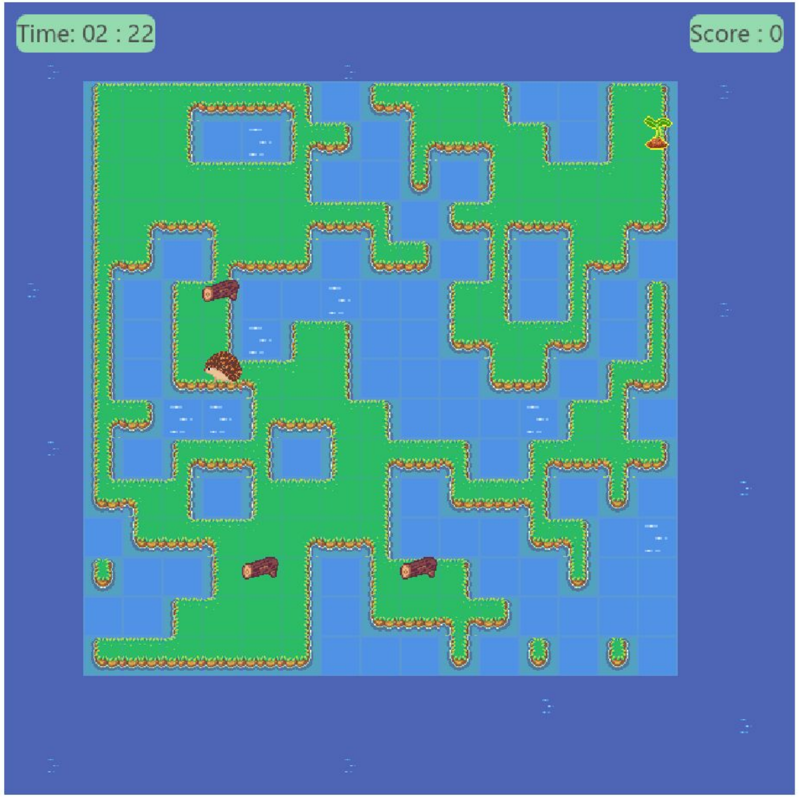
About us & Promotion

- Jakob Karalus - PhD Student at AI Institute - Human-in-the-Loop Reinforcement Learning & Explainability
- Conny Olz - Postdoc at AI Institute - Hierarchical Symbolic Planning

Self-Promotion: **Project “AI in Games”**

- Masters project at Ulm University
- Offered by the Institute of Artificial Intelligence
- Every semester, this summer term Tuesdays 14-16 pm
- Task: Develop a video game as a group using AI methods
    - Very similar to this GameJam

Self-Promotion: Project “AI in Games”



Self-Promotion: Project "AI in Games"

Outline

- Level Generation
  - Wave Function Collapse
  - Constraint Satisfaction Problem (CSP)
- Agent Movement
  - Pathfinding recap
  - Group movement with Boids
- Agent Action Selection
  - A* Search
  - Minimax

Generation with Wave Function Collapse

Wave Function Collapse - Idea

- Given:
  - An empty grid
  - A set of possible tiles
  - A set of constraints for the tiles.
- Example:
  - Water can be near water or beach (Blue)
  - Beach can be near beach, water or grass
  - Grass can be near beach, grass or hills
  - Hills can be near grass or hills
- Algorithms

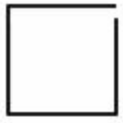
Init grid with each value possible for each tile.

While not all tiles filled:

> *Select tile with lowest entropy and collapse it (if multiple, select randomly)*
>
> *Propage constraints*

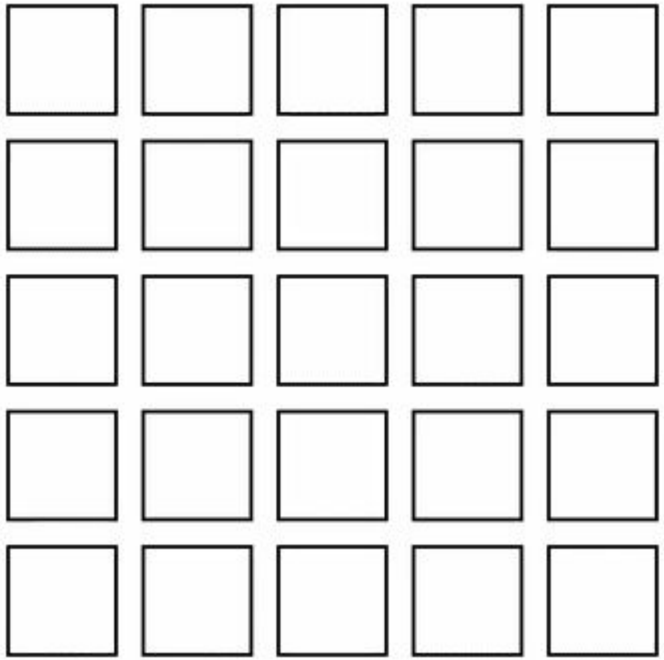
WFC - Example

Initialize empty grid



WFC - Example

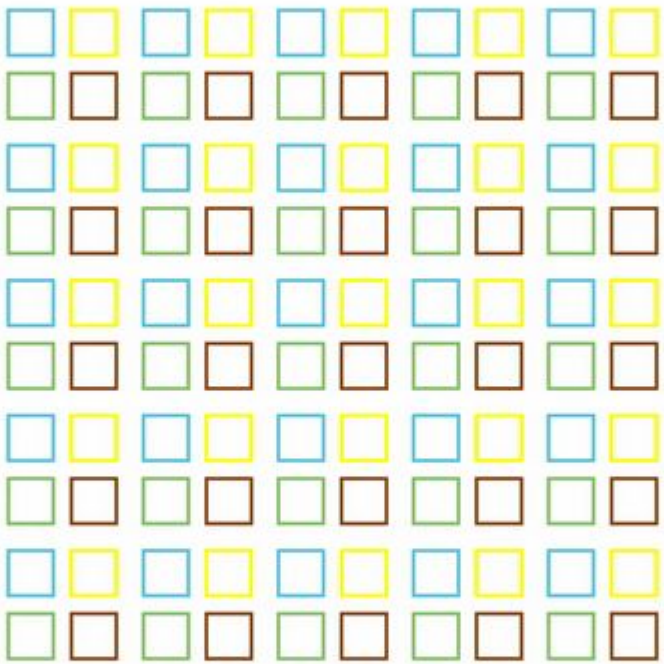
Initialize every tile with all possible values (sometimes called waves)



WFC - Example

Select the lowest entropy cell -> All cell the same, so random.

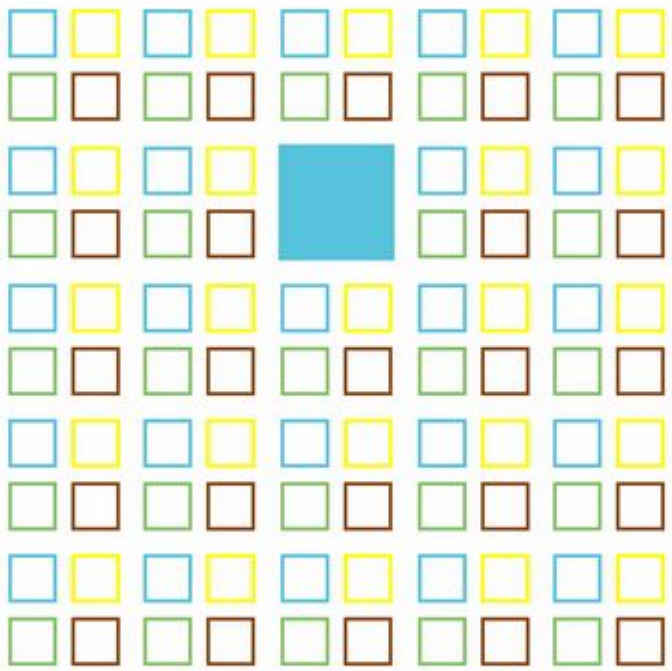
Collapse the cell to one (still valid) value.



WFC - Example

Propagate constraints after collapsing.

Only water and beach can be near water.

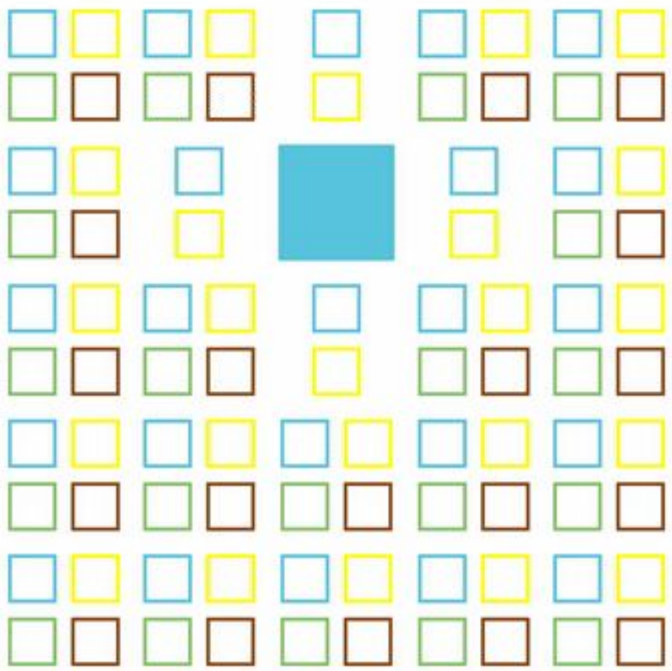


WFC - Example

Propagate constraints after collapsing.

Since Hills can be only near water, a further set of tills can have some constraints collapsed.

Collapse constraints till no changes occur.

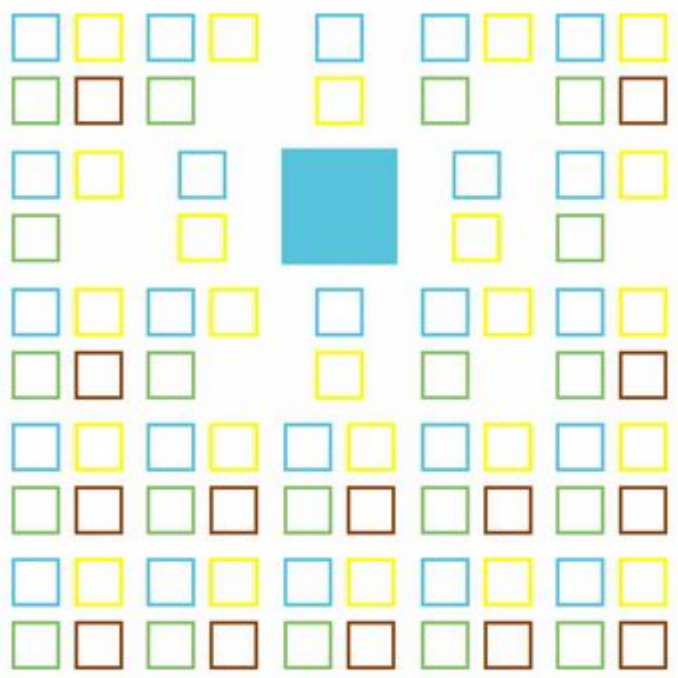


WFC - Example

Select the lowest entropy cell -> One of the 4 cells with only 2 constraints left.

Collapse the cell to one (still valid) value.

Only water/beach left, so 50/50 chance.



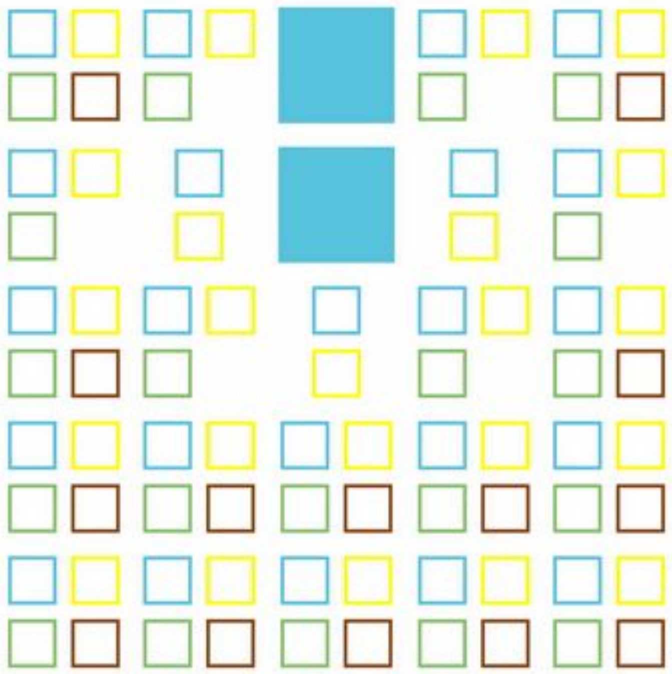
WFC - Example

Propagate constraints after collapsing.

Only water and beach can be near water.

Hills can only be near grass.

Collapse constraints till no changes occur.



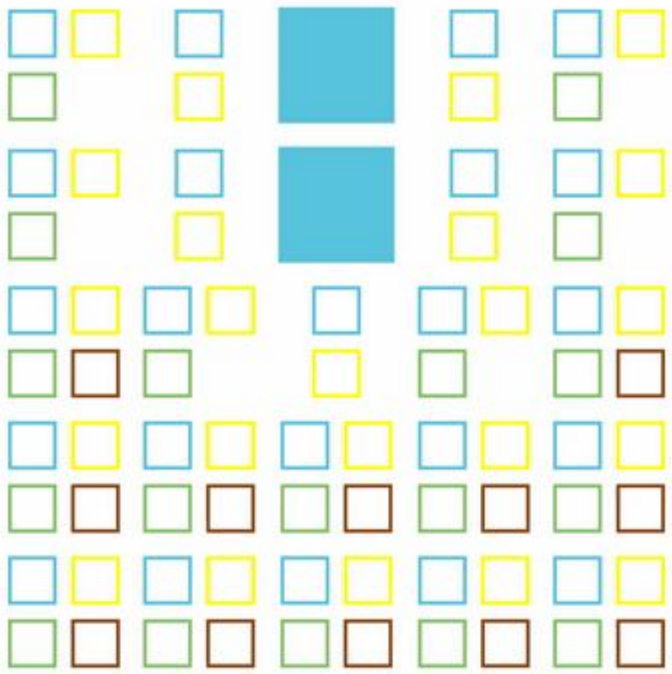
WFC - Example

Select the lowest entropy cell.

Collapse the cell to one (still valid) value.

Only water/beach left, so 50/50 chance.

No propagation of constraints needed.

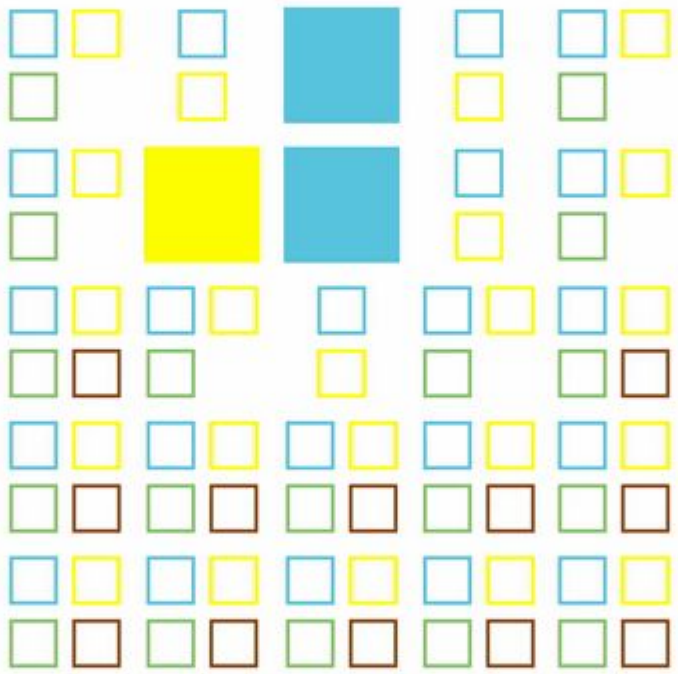


WFC - Example

Select the lowest entropy cell.

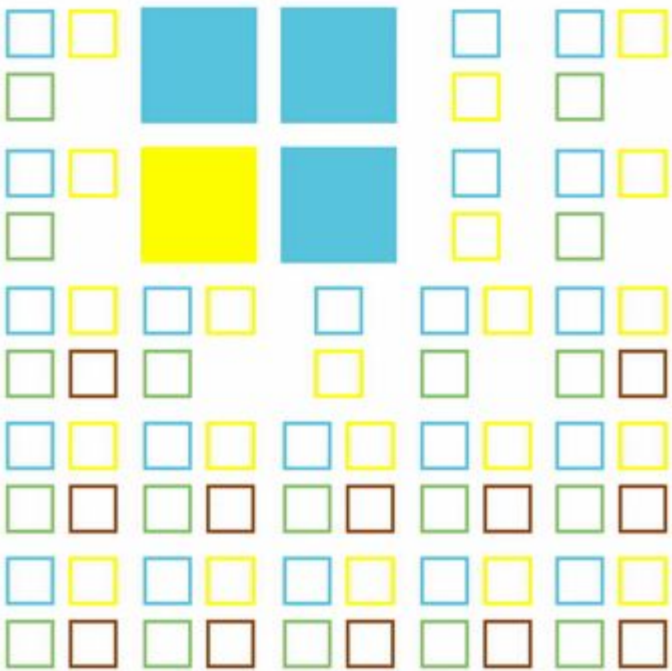
Collapse the cell to one (still valid) value.

Only water/beach left, so 50/50 chance.



WFC - Example

Update constraints.

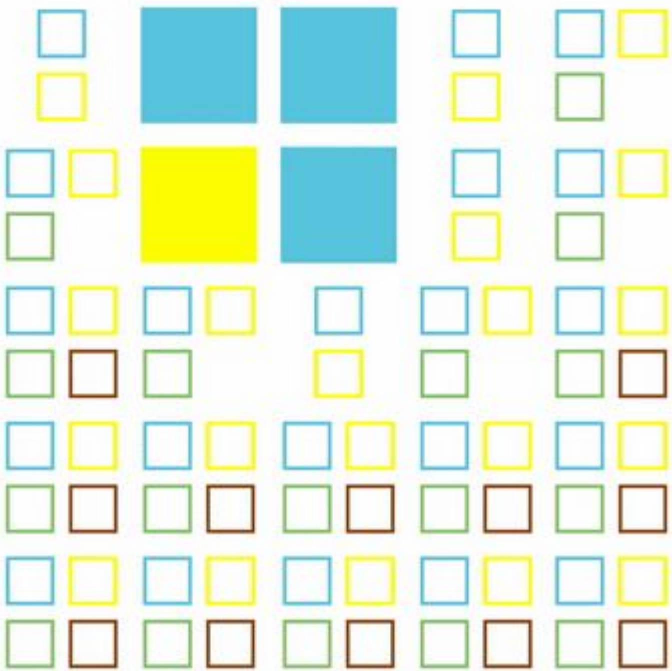


WFC - Example

Select the lowest entropy cell.

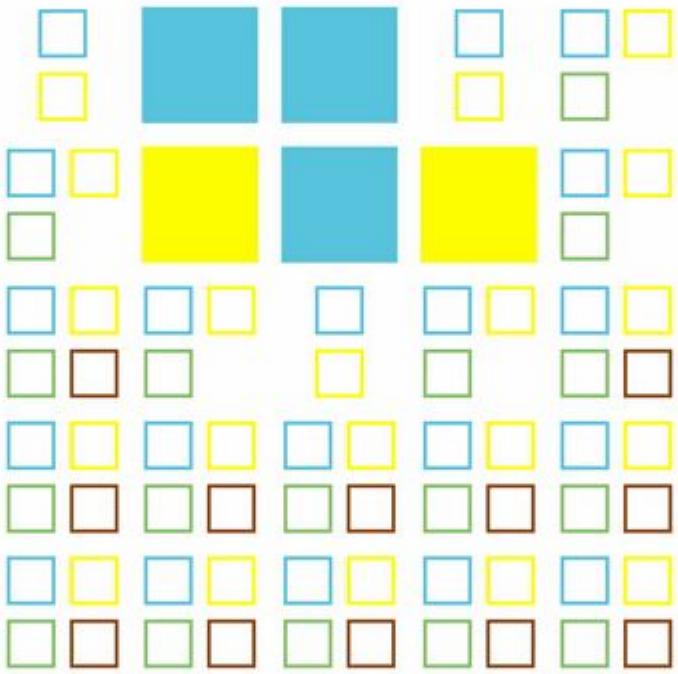
Collapse the cell to one (still valid) value.

Only water/beach left, so 50/50 chance.



WFC - Example

Update constraints

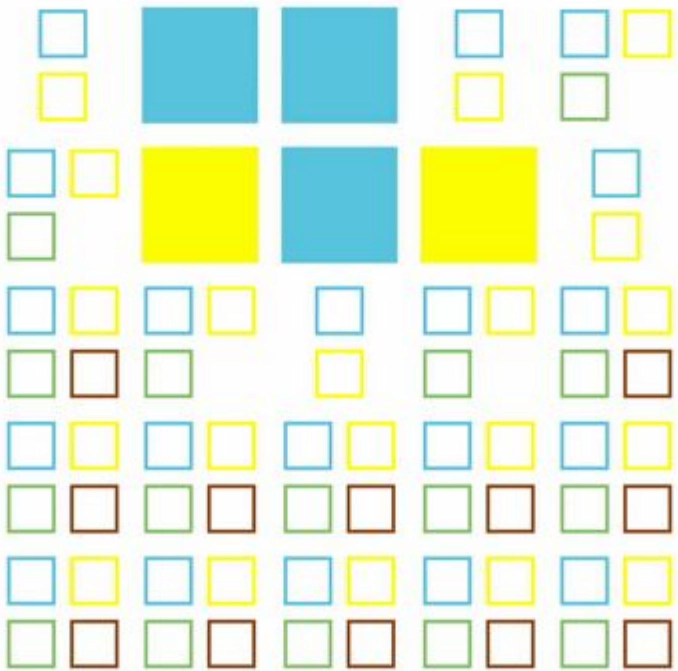


WFC - Example

Select the lowest entropy cell.

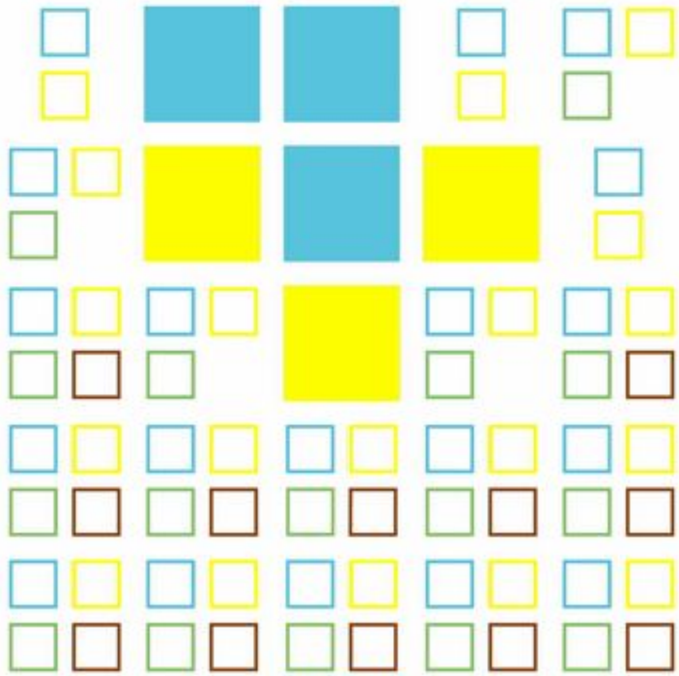
Collapse the cell to one (still valid) value.

Only water/beach left, so 50/50 chance.



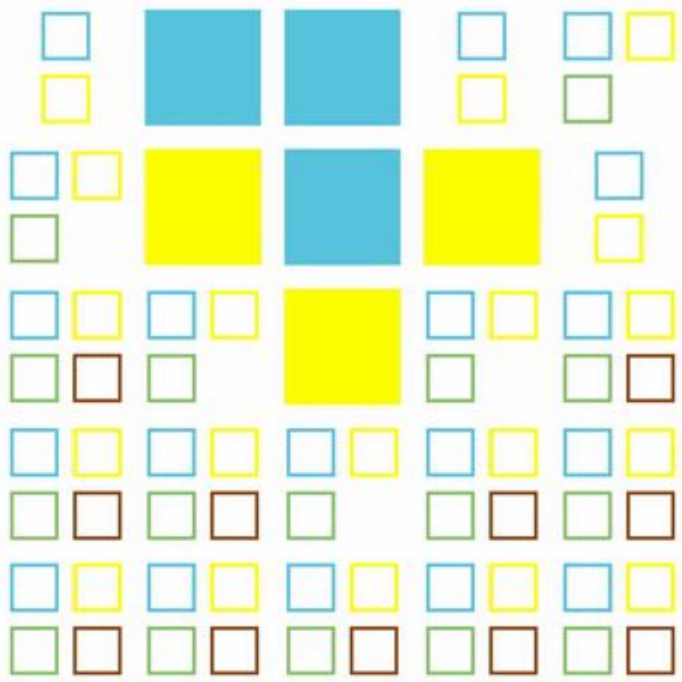
WFC - Example

Update constraints



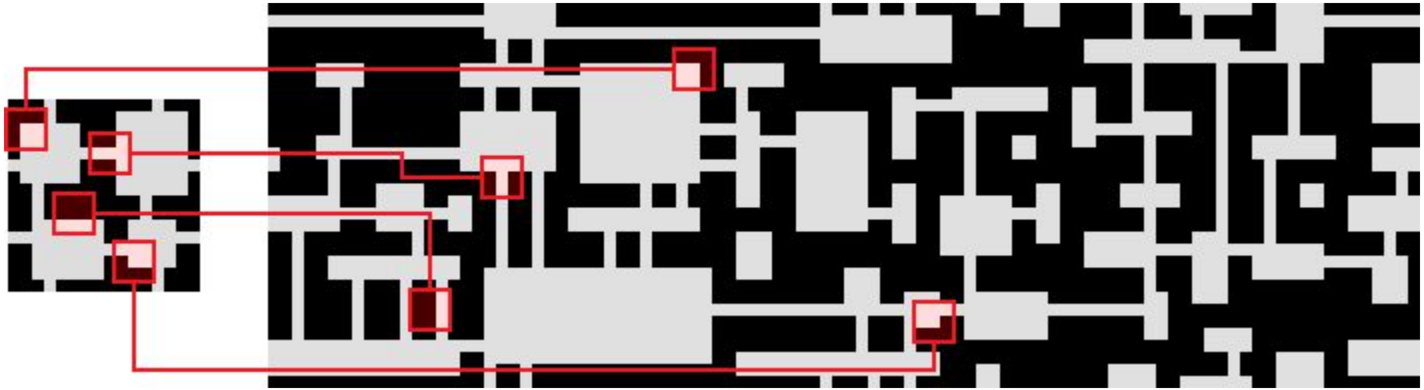
WFC - Example

Initialize empty grid



WFC - How to write constraints

- Writing all constraints can be tedious.
- Idea: Generate a example level which contains possibles allowed connections of tiles
- All NxN pattern in the output have to be in the input.
- Optional: The distribution of pattern should similar in the input and output pattern.
- Eg: https://github.com/mxgmn/WaveFunctionCollapse



WFC Online Demo

https://oskarstalberg.com/game/wave/wave.html

WFC Advantages

Advantages:

- WFC is quite easy to understand and simple to implement
- WFC can work with existing tiles (aka handplace a part of tiles)
- The constraints interface through example maps are quite convenient

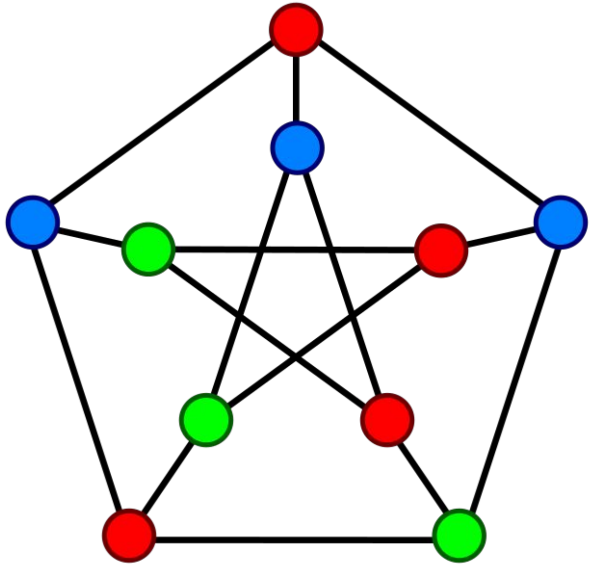
Problems:

- Large states spaces
  - Multiple rooms in our dungeon crawler
- Constraints over a large amount of tiles
  - “There should exists a path between all rooms”
- Hard to specify example maps for large and different configuration
  - I want to have rooms with multiple biomes
- Abstract constraints
  - The player should not see the same monster type twice in a row

Constraints Satisfaction Problems

CSP

- Wave Function Collapse is just a special version of the “Constraint Satisfaction Problem”:
- CSP have
  - A set of states S - *2D Grid for the WFC, CSP could be any shape*
  - A set of variables - *Possible tile values in WFC, here similar*
  - A set of constraints -
- One of the common textbook problems of CSPs is the Graph Coloring problems



Example 2: Sudoku

- Variables: X_11, ..., X_99, one variable for every cell
- Domains: {1, ..., 9}
- Constraints:
  - Row constraints: X_11 ≠ X_12, ... , X_11 ≠ X_19, ...
  - Column constraints: X_11 ≠ X_21, ... , X_11 ≠ X_91, ...
  - Block constraints: X_11 ≠ X_12, ... , X_11 ≠ X_33, ...

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Example 3: Cryptoarithmetic Problem

- Variables: S, E, N, D, M, O, R, Y
- Domains: {0, … , 9}
- Constraints:
  - $S \neq E \neq N \neq D \neq M \neq O \neq R \neq Y$
  - $D + E = Y + 10*C1$
  - $C1 + N + R = E + 10*C10$
  - $C10 + E + O = N + 10*C100$
  - $C100 + S + M = O + 10* C1000$
  - $C1000 = M$

  - (C1 … C1000 are auxiliary variables representing the digit carried over)

$$
\begin{array}{ccccc}
 & S & E & N & D \\
+ & M & O & R & E \\
\hline
M & O & N & E & Y
\end{array}
$$

CSP Solving - Backtracking

Example:

1. A must be different from B.
2. B must be greater than C.
3. Possible values for A,B & C are {1,2,3}

- Start with an empty assignment: `{}`.
- Start with variable A and value 1.
- Check if the assignment satisfies all constraints:
  - Since A is not yet assigned, the first constraint is trivially satisfied.
  - There are no other variables assigned, so the second constraint is trivially satisfied.
- Recursively assign values to the next variable (B).
- Choose variable B and value 1.
- Check if the assignment satisfies all constraints:
  - A is assigned 1, so the first constraint is violated.
  - There are no other variables assigned, so the second constraint is trivially satisfied.
- Backtrack and try a different value for B.
- Choose variable B and value 2.
- Check if the assignment satisfies all constraints:
  - A is assigned 1, so the first constraint is satisfied.
  - There are no other variables assigned, so the second constraint is trivially satisfied.
- Recursively assign values to the next variable (C).
- Choose variable C and value 1.
- Check if the assignment satisfies all constraints:
  - A is assigned 1, so the first constraint is satisfied.
  - B is assigned 2, so the second constraint is satisfied.
- All variables are assigned values, so we have found a solution: `{A: 1, B: 2, C: 1}`.

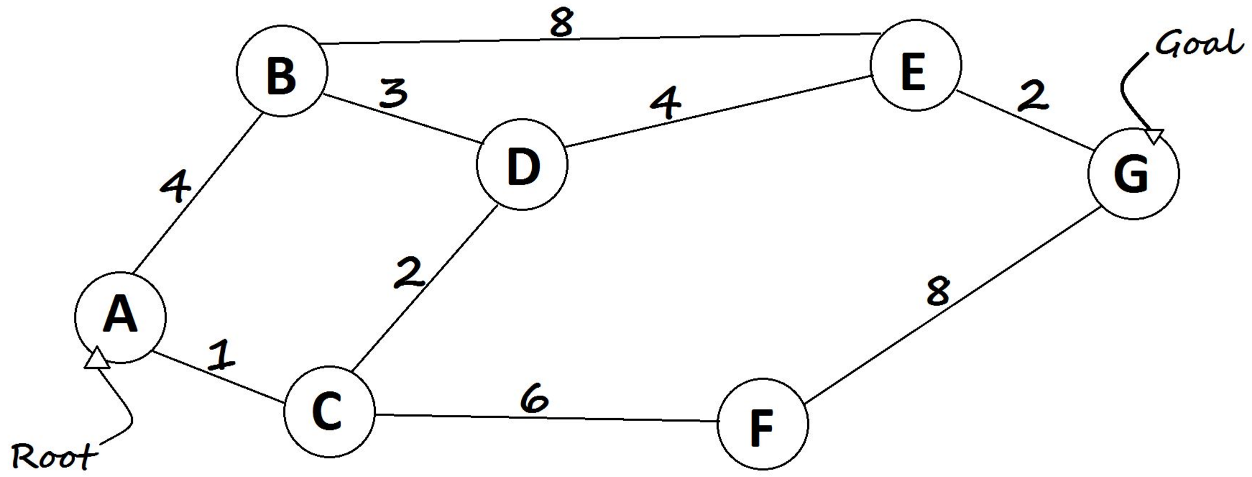
WFC & CSP Summary

- CSP contain a number of more sophisticated solving technique, which allow for much more complicated constraint setup.
    - While techniques like backtracking, constraint propagation (forward checking) are simple, efficient implementation requires a bit of care
- In general WFC is better/faster in domains where a lot of solutions exists and require only local constraints. In settings where just a few solutions exists, classical CSP often outperform then.
- Classical CSP solver are a great complement to WFC style algorithms
    - E.g. Generate high-level graphs of room structure/biomes/enemies in a CSP. Low level tiling with WFC
- Software:
    - MiniZinc: https://www.minizinc.org/ powerful CSP solver
    - Google OR Tools: https://developers.google.com/optimization collection of various optimization solvers
        - Kapsack problems - Fit the most item into a give space
        - Sheduling problems - Assign units to workspace to archive the highest utility
        - Traveling salesman - Shortest route to visit every location
        - ...

Agent Movement

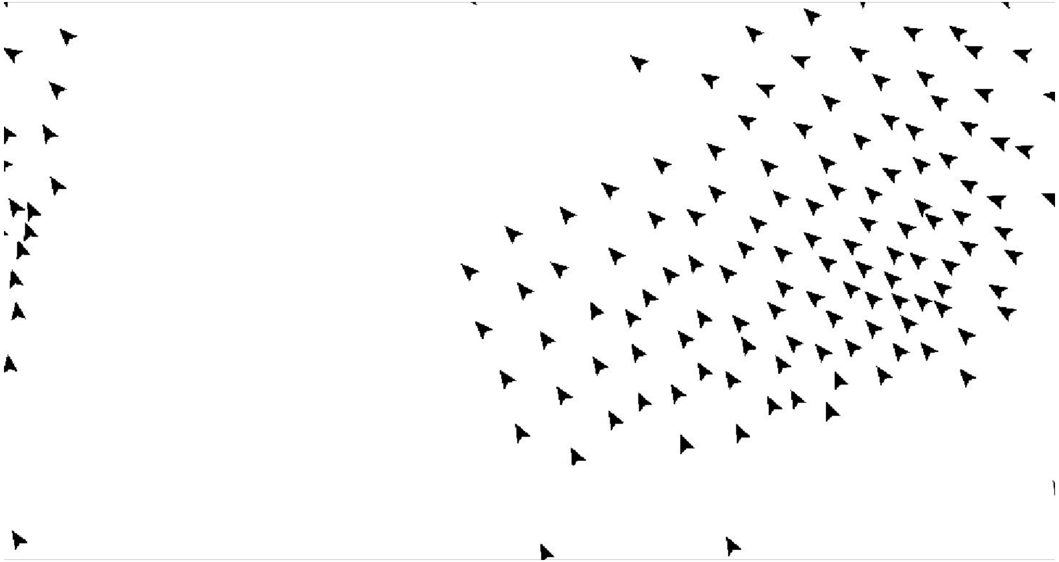
Path finding

- Short recap A* and Djikstra
- Probably already implemented in many game engines



Multiple Agents - Boids

- Individual path finding with possible overlapping routes can be computation expensive.
- Naive approach of just planning and then replanning leads to a lot of stuttering
- Boids:
  - Treat agents as a mass with the center moving into the right direction.



Boids - Pseudocode

Each entity is called a boid, with a position and a velocity (includes direction here)

Behaviour of boids is controlled by three factors:

- Separation
- Alignment
- Cohesion

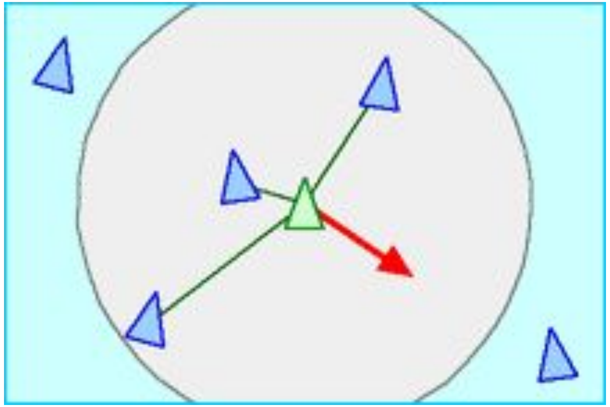
```
def update_boids(boids):
    """Update boids based on separation, alignment, and cohesion."""
    for i, boid in enumerate(boids):
        separation_force = separation(boids, i, SEPARATION_DISTANCE)
        alignment_force = alignment(boids, i, ALIGNMENT_DISTANCE)
        cohesion_force = cohesion(boids, i, COHESION_DISTANCE)

        # Apply forces
        boid.acceleration += separation_force * SEPARATION_WEIGHT
        boid.acceleration += alignment_force * ALIGNMENT_WEIGHT
        boid.acceleration += cohesion_force * COHESION_WEIGHT

        # Update velocity and position
        boid.velocity += boid.acceleration
        boid.velocity.limit(MAX_SPEED)
        boid.position += boid.velocity
        boid.acceleration *= 0  # Reset acceleration
```

Boids - Pseudocode Separation

Each boid tries to keep a reasonable distance
to all other boids in the same cluster.

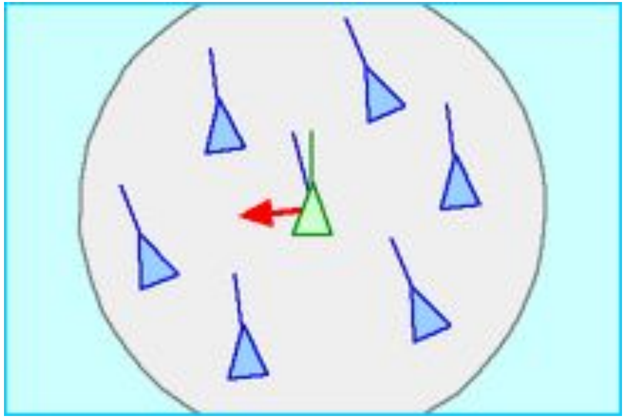


```
def separation(boids, boid_index, separation_distance):
    """Calculate separation for a single boid."""
    steering = Vector(0, 0)
    count = 0
    for other_boid in boids:
        if other_boid is not boids[boid_index]:
            distance = distance_between(boids[boid_index].position, other_boid.position)
            if distance < separation_distance:
                diff = boids[boid_index].position - other_boid.position
                diff.normalize()
                diff /= distance
                steering += diff
                count += 1
    if count > 0:
        steering /= count
    if steering.length() > 0:
        steering.normalize()
        steering *= MAX_SPEED
        steering -= boids[boid_index].velocity
        steering.limit(MAX_FORCE)
    return steering
```

Boids - Pseudocode - Alignment

Each boid tries to align itself into the same direction than all other boids in his cluster.

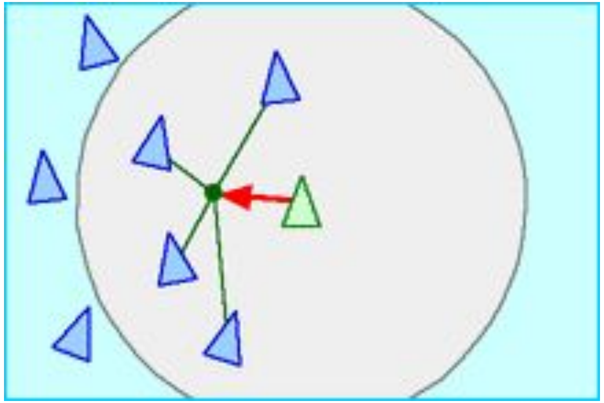
```
def alignment(boids, boid_index, alignment_distance):
    """Calculate alignment for a single boid."""
    steering = Vector(0, 0)
    count = 0
    for other_boid in boids:
        if other_boid is not boids[boid_index]:
            distance = distance_between(boids[boid_index].position, other_boid.position)
            if distance < alignment_distance:
                steering += other_boid.velocity
                count += 1
    if count > 0:
        steering /= count
        steering.normalize()
        steering *= MAX_SPEED
        steering -= boids[boid_index].velocity
        steering.limit(MAX_FORCE)
    return steering
```



Boids - Pseudocode - Cohesion

Each boids tries to move into the direction of the mean positions of all other boids (in his cluster).

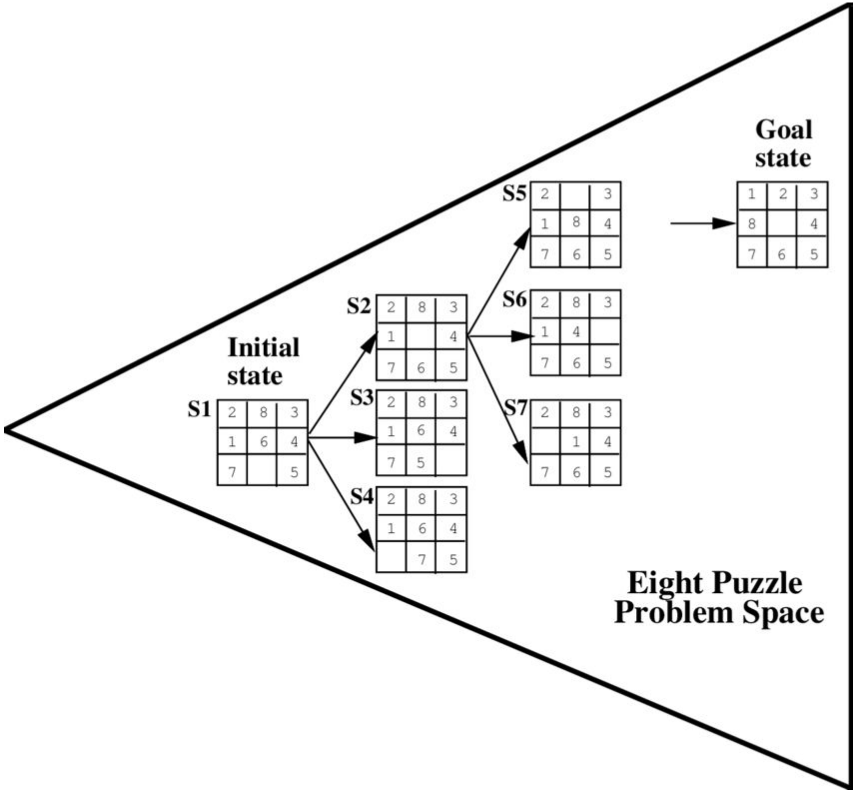
```
def cohesion(boids, boid_index, cohesion_distance):
    """Calculate cohesion for a single boid."""
    steering = Vector(0, 0)
    count = 0
    for other_boid in boids:
        if other_boid is not boids[boid_index]:
            distance = distance_between(boids[boid_index].position, other_boid.position)
            if distance < cohesion_distance:
                steering += other_boid.position
                count += 1
    if count > 0:
        steering /= count
        steering -= boids[boid_index].position
        steering.normalize()
        steering *= MAX_SPEED
        steering -= boids[boid_index].velocity
        steering.limit(MAX_FORCE)
    return steering
```



Agent Action Selection

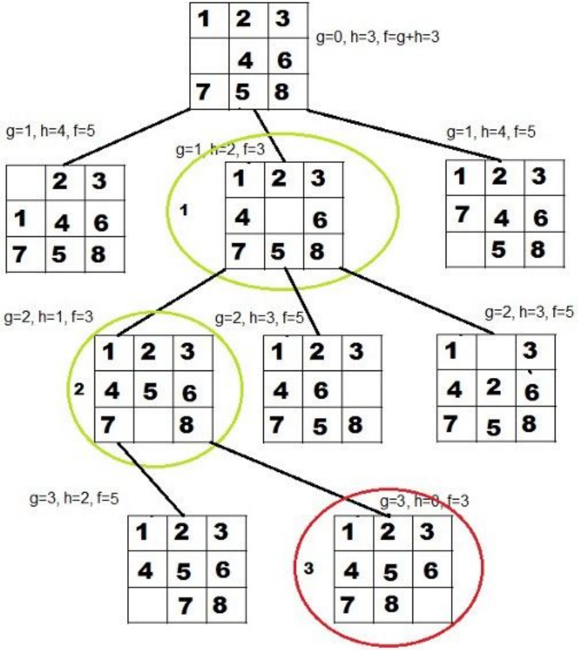
Search-based Problems

- Represent the problem as tree/graph
  - States are nodes
  - Actions are edges
- Search through the tree for the best option
- Whole field of symbolic planning:
  - Idea: Just model the world, its states and actions and let the symbolic AI systems find the a action plan from the start to the goal
  - Requires a bit of reading, but can lead to really adaptive agents
  - See https://www.fast-downward.org/



A* - Search with a Heuristic

- Given a state/node, a heuristic estimates how many actions are still necessary to reach a goal state
  - E.g., count number of correct tiles
- A*: Expand node n with lowest f-value
  - f(n) = g(n) + h(n)
  - g(n): costs to reach n
  - h(n): heuristic value of n
- Guarantees to find optimal solution if heuristic fulfills certain properties (admissible)

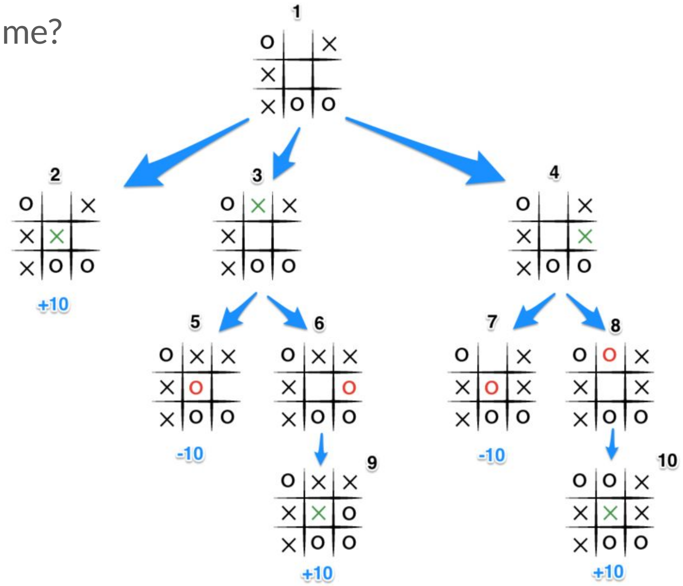


Search - Multiple Players

Minimax

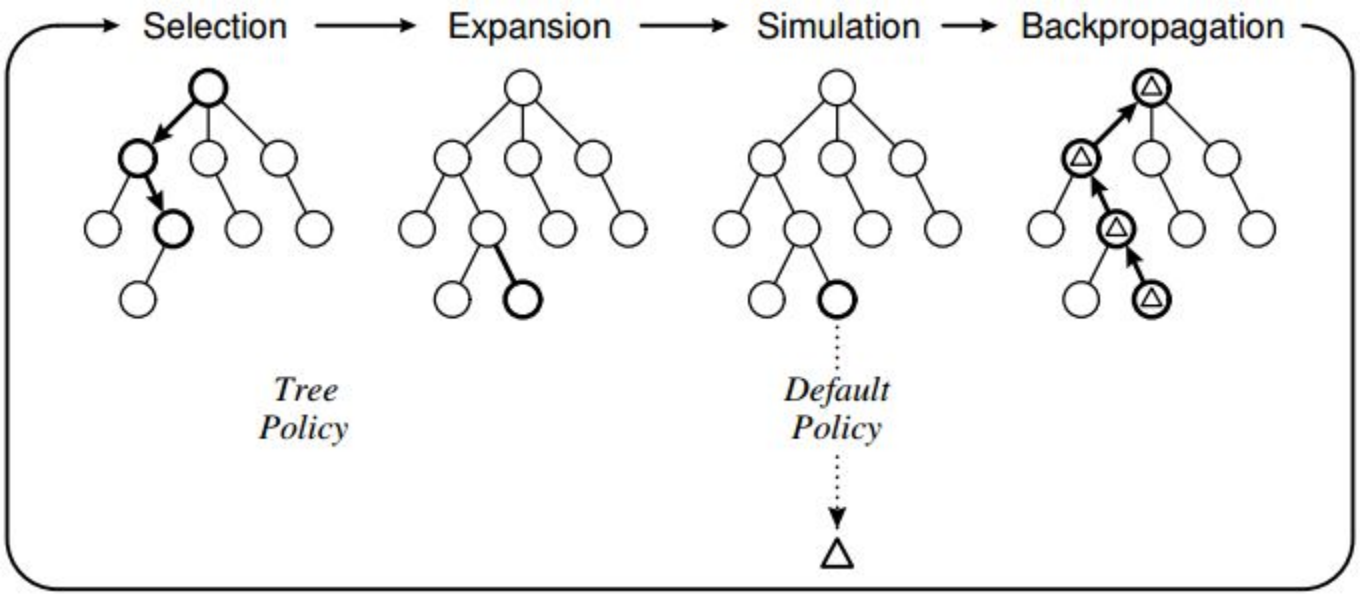
- 2-Player games
- Assumptions: Fully observable, discrete, opponent plays optimal
- Question: What's the best move/strategy for winning the game?
- Solution:
  - Model the game as a search tree
  - Propagate results upwards
- Performance improvement: Alpha-Beta (pruning)

- Limitation: Large state spaces
  -> Monte Carlo Tree Search (next slides)



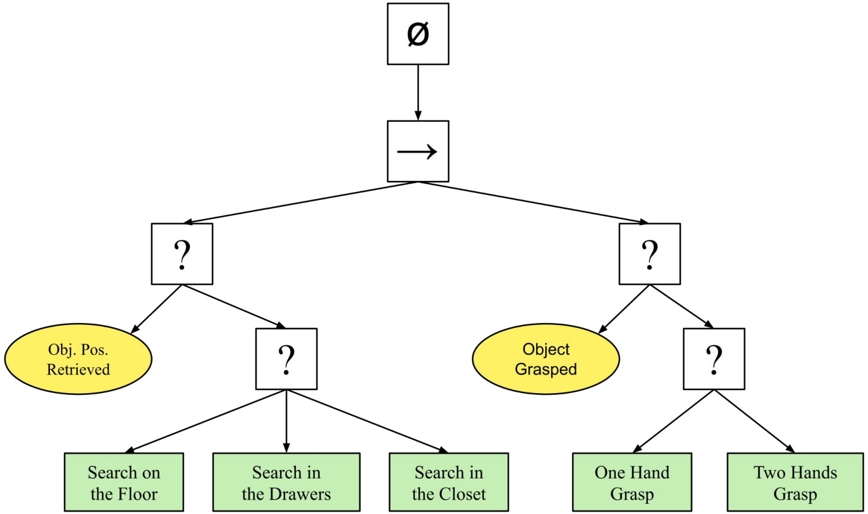
Very large search spaces

- If the search space get extremely large, sampling is needed.
- Probably waaaaay out of scope for this talk.



Behaviour Tree's

- For most games sophisticated search based action selection is probably overkill
- "Dumb Agents" doesn't necessary mean boring gameplay
  - Dark Souls have on the most scripted/simple reactive enemies...
- Direct often ascripting viable solution
- Behaviour Trees provide a bit more structure (especially in larger projects)
  - Combination between decisions trees and state machines



Realistic action selection - Thompson Sampling

- Always selecting the best option can lead to repetitive agents
- Random actions can lead to “stupid” agents
- Inspiration:Thompson Sampling:
  - Select an action proportional to how good it is
  - From the Multi-Armed-Bandit setting, but can be applied to any situation, either in search or direct action selection.
- How do we get to probabilities?
  - Softmax function
  - Can be adjusted through the temperature

```python
import numpy as np
def softmax(logits, temperature=1):
  """
  Applies the softmax function to a list of logits.

  Args:
      logits: A list of floats representing unnormalized log probabilities.
      temperature: A float (default: 1) to control the "peakedness" of the distribution.
          Higher temperature leads to a more uniform distribution.

  Returns:
      A list of floats representing the normalized probabilities.
  """
  e = np.exp(np.array(logits) / temperature)
  return e / np.sum(e)

def sample(probs):
  """
  Samples an index from a probability distribution.

  Args:
      probs: A list of floats representing probabilities.

  Returns:
      An integer representing the sampled index.
  """
  return np.random.choice(len(probs), p=probs)

# Example usage
logits = [1.0, 5.0, -0.1]
print(softmax(logits))  # Output: [0.01787917 0.97616938 0.00595146]
print(softmax(logits, temperature=3)) # Output: [0.18225863 0.69142873 0.12631264]]
sample_index = sample(softmax(logits))
print(sample_index)  # Output: An integer between 0 and 2 (inclusive
```

Final notes

- Methods can be easily nested. Don't try to do everything with one method.
- Realistically for short project, hacking something together is often fine. For larger problems the usage of established tools can be highly beneficial.
-

Questions?
jakob.karalus@uni-ulm.de