COMP250: Artificial Intelligence

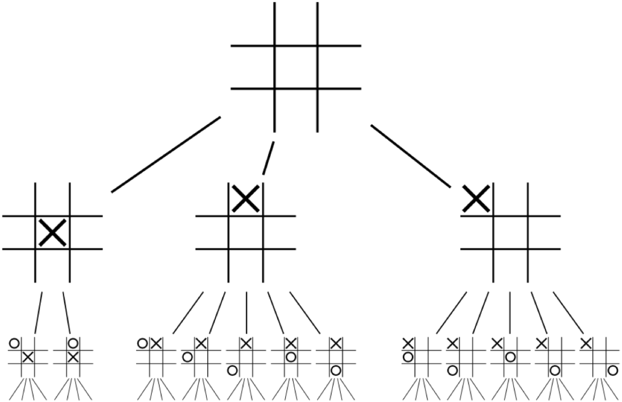
4: Minimax Search

Noughts and Crosses

- Clone the following repository: `https://github.com/Falmouth-Games-Academy/comp250-live-coding`
- Open `COMP250/04_minimax` in PyCharm and run `oxo_main.py`
- If PyCharm asks for license server information, enter `http://trlicefal.fal.ac.uk`

Minimax search

Game trees



Minimax

- Terminal game states have a **value**
  - E.g. $+1$ for a win, -1 for a loss, 0 for a draw
- I want to **maximise** the value
- My opponent wants to **minimise** the value
- Therefore I want to **maximise** the **minimum** value my opponent can achieve
- This is generally only true for **two-player zero-sum** games

Minimax search

- Recursively defines a **value** for non-terminal game states
- Consider each possible “next state”, i.e. each possible move
- If it’s my turn, the value is the **maximum** value over next states
- If it’s my opponent’s turn, the value is the **minimum** value over next states

Minimax search pseudocode

```
procedure MINIMAX(state, currentPlayer)
    if state is terminal then
        return value of state
    else if currentPlayer = 1 then
        bestValue = −∞
        for each possible nextState do
            v = MINIMAX(nextState, 3− currentPlayer)
            bestValue = MAX(bestValue, v)
        end for
        return bestValue
    else if currentPlayer = 2 then
        bestValue = +∞
        for each possible nextState do
            v = MINIMAX(nextState, 3− currentPlayer)
            bestValue = MIN(bestValue, v)
        end for
        return bestValue
    end if
end procedure
```

Stopping early

for each possible nextState **do**
    v = MINIMAX(nextState, 3− currentPlayer)
    bestValue = MAX(bestValue, v)
end for

- State values are always between -1 and $+1$
- So if we ever have bestValue $= 1$, we can stop early
- Similarly when minimising if bestValue $= -1$

Using minimax search

- To decide what move to play next...
- Calculate the minimax value for each move
- Choose the move with the maximum score
- If there are several with the same score, choose one at random

Minimax and game theory

- For a **two-player zero-sum** game with **perfect information** and **sequential moves**
- Minimax search will always find a **Nash equilibrium**
- I.e. a minimax player plays **perfectly**
- **But...**

Heuristics for search

Minimax for larger games

- The game tree for noughts and crosses has only a few thousand states
- Most games are too large to search fully
  - Connect 4 has $\approx 10^{13}$ states
  - Chess has $\approx 10^{47}$ states

Depth limiting

- Standard minimax needs to search all the way to **terminal** (game over) states
- **Depth limiting** is a common technique to apply minimax to larger games
- Still evaluate terminal states as $+1$ / 0 / -1
- For nonterminal states at depth d, apply a heuristic evaluation instead of searching deeper
- Evaluation is a number between -1 and $+1$, estimating the probable outcome of the game

1-ply search

- Case $d = 1$
- For each move, evaluate the state resulting from playing that move
- This is computationally fast
- Often easier to design a “which state is better” heuristic than to directly design a “which move to play” heuristic

Move ordering

- Minimax can **stop early** if it sees a value of $+1$ for maximising player or -1 for minimising player
- Modifications to minimax algorithm (e.g. **alpha-beta pruning**) lead to more of this
- Thus ordering moves from **best to worst** means faster search
- How do we know which moves are "best" and "worst"? Use a heuristic!

Designing heuristics

- The **playing strength** of depth limited minimax depends heavily on the design of the **heuristic**
- Good heuristic design requires **in-depth knowledge** of the tactics and strategy of the game
- Next time we will look at what we can do if we don't possess such knowledge

Planning

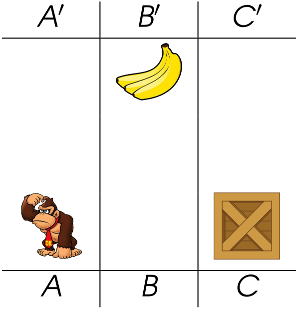
Planning

- An **agent** in an **environment**
- The environment has a **state**
- The agent can perform **actions** to change the state
- The agent wants to change the state so as to achieve a **goal**
- Problem: find a sequence of actions that leads to the goal

STRIPS planning

- **St**anford **R**esearch **I**nstitute **P**roblem **S**olver
- Describes the state of the environment by a set of **predicates** which are true
- Models a problem as:
  - The **initial state** (a set of predicates which are true)
  - The **goal state** (a set of predicates, specifying whether each should be true or false)
  - The set of **actions**, each specifying:
    - Preconditions (a set of predicates which must be satisfied for this action to be possible)
    - Postconditions (specifying what predicates are made true or false by this action)

STRIPS example



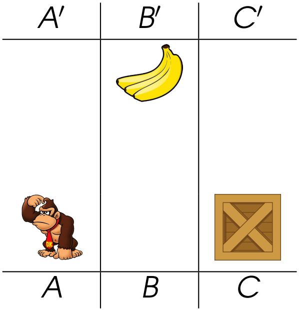
Initial state:

```
At(A),
BoxAt(C),
BananasAt(B')
```

Goal:

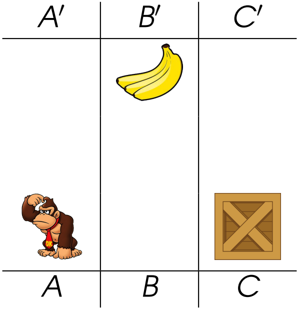
```
HasBananas
```

STRIPS example — Actions

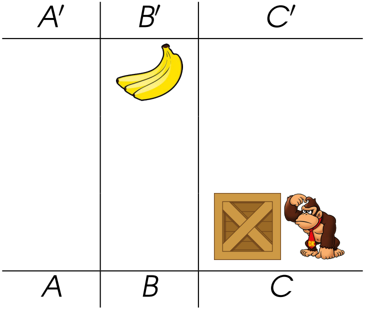



```
Move(x, y)
  Pre:  At(x)
  Post: !At(x), At(y)

ClimbUp(x)
  Pre:  At(x), BoxAt(x)
  Post: !At(x), At(x')

ClimbDown(x')
  Pre:  At(x'), BoxAt(x)
  Post: !At(x'), At(x)

PushBox(x, y)
  Pre:  At(x), BoxAt(x)
  Post: !At(x), At(y),
        !BoxAt(x), BoxAt(y)

TakeBananas(x)
  Pre:  At(x), BananasAt(x)
  Post: !BananasAt(x), HasBananas
```

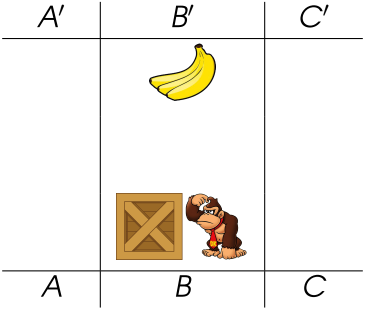
STRIPS example — Solution



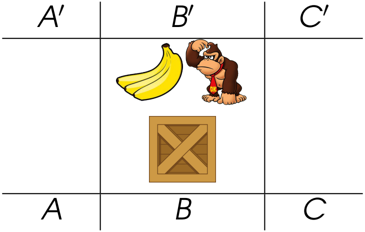
STRIPS example — Solution



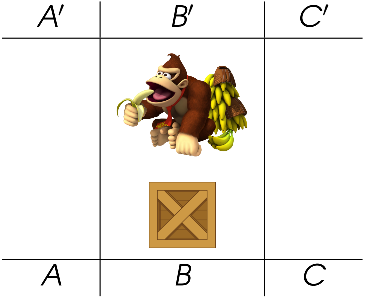
STRIPS example — Solution



STRIPS example — Solution



STRIPS example — Solution



Finding the solution

- For a given state, we can construct a list of all **valid actions** based on their **preconditions**
- We can also find the **next state** resulting from each action based on their **postconditions**
- This should sound familiar (from 2 weeks ago)...
- We can construct a **tree** of states and actions
- We can then **search** this tree to find a goal state

Tree traversal

```
procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty do
        pop n from S
        push children of n onto S
    end while
end procedure

procedure BreadthFirstSearch
    let Q be a queue
    enqueue root node into Q
    while Q is not empty do
        dequeue n from Q
        enqueue children of n into Q
    end while
end procedure
```

Tree traversal example