Boop - AI Agent

Alexander Barry, Scott Berlow, Christopher Mascis

Problem Statement and Analysis

- Design an AI Agent to play *boop*
- Evaluate several AI algorithms
- Create a user interface to challenge the AI
- Implement using Python

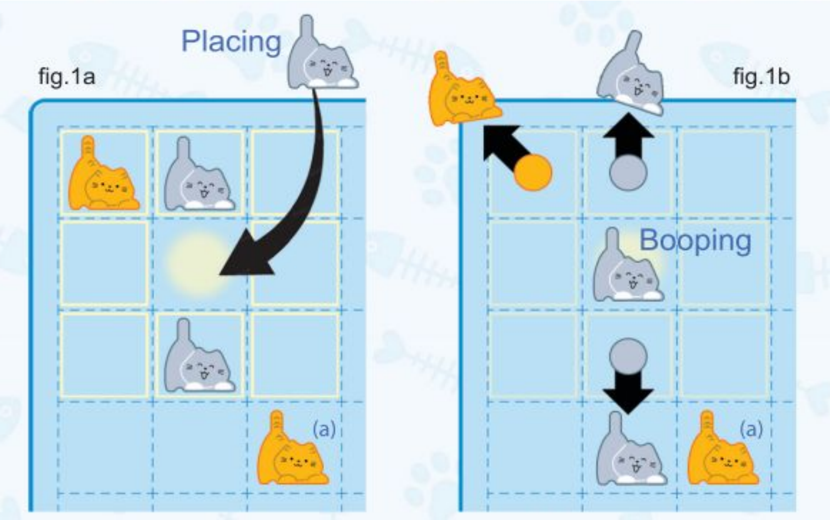
Smirk & Dagger Games

Characteristics of *boop*

- Two-player
- Zero-sum
- Fully Observable
- Turn-taking
- Multiple Win Conditions

General Gameplay

Booping kittens

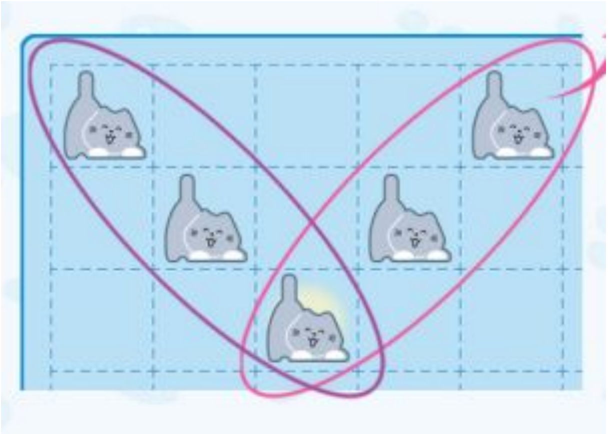


Graduating kittens

From the *boop* Rulebook

Additional Rules

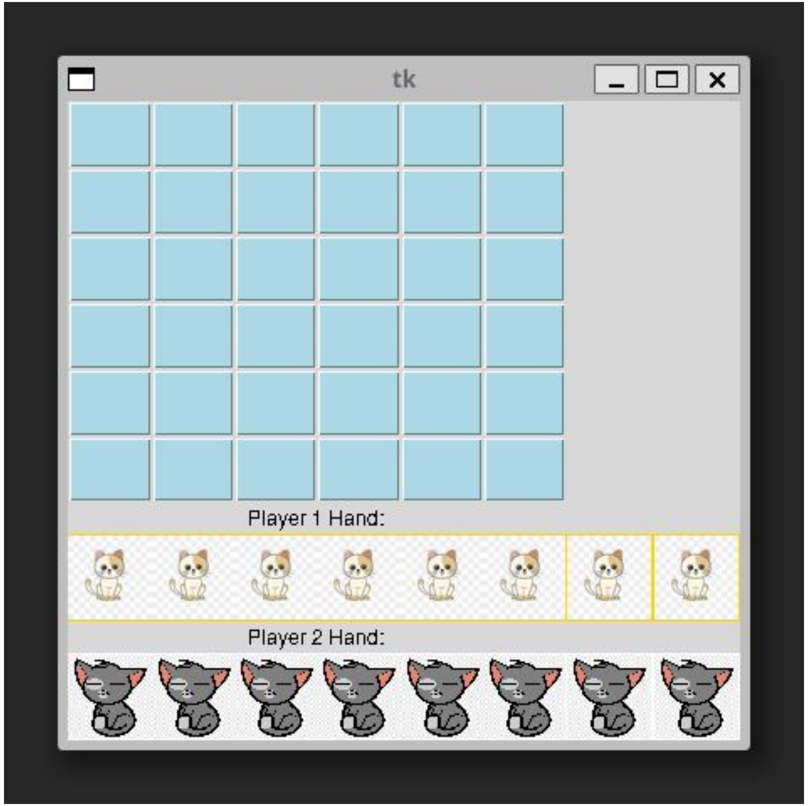
- Multiple "triples" in one turn
  - Player gets to decide which kittens to promote
- Triples for both players in one turn
  - Each player gets to promote
- All eight pieces on the board
  - Player chooses one to promote



From the *boop* Rulebook

Use-Case Scenarios

- Human vs Human:
  - play the board game in a virtual UI against another human
- Human vs AI:
  - play without another person present
  - practice against an AI to hone skills
- AI vs AI:
  - self-play mode
  - evaluate agent types against each other
  - evaluate heuristic functions and potential features
  - explore strategies



State Representation

```
GameState
├── Player1
│   ├── Hand
│   │   ├── Kittens[]
│   │   └── Cats[]
│   ├── Decisions[]
│   └── Agent?
├── Player2
│   └── (same)
├── Board
│   └── Grid[6][6]
├── Turn
├── Must_promote
└── Winner
```

```
MinimaxAgent picked (K(2), 2, 3) with value
-2.65, states explored: 12097

AI action: K(2) to (2, 3)

  |------|------|------|------|------|------|
0 |      |      |      |      |      | K(2) |
  |------|------|------|------|------|------|
1 | K(1) |      |      |      |      |      |
  |------|------|------|------|------|------|
2 |      |      |      | K(2) |      |      |
  |------|------|------|------|------|------|
3 |      |      |      |      |      |      |
  |------|------|------|------|------|------|
4 |      | K(1) |      |      |      |      |
  |------|------|------|------|------|------|
5 |      |      |      |      |      |      |
  |------|------|------|------|------|------|
     0      1      2      3      4      5
```

AI Algorithm and Model

- Minimax
  - Configurable depth (where depth accounts for a layer of MAX and MIN each)
  - **Problem: too many states!**
  - Worst case: `~36^(d*2)` possible *spaces* to consider
    - If a player has cats and kittens, 2 possible moves *per space* at each level
  - Initial move with depth 2: `~36^4 = 1,679,616` states
  - With cats: `~72^4 = 26,873,856` states
  - Configurable `maxStates` parameter to cap exploration - but leads to suboptimal moves
- Alpha-Beta Pruning
  - Greatly reduces overall computation, though still can be slow
- Beam Search
  - Configurable beam "width" parameter
  - Forward pruning: at each level, maintain only the best *n* states, where *n* is the beam width
  - Shannon's Type B strategy: narrow but deep search

Evaluation Functions

- eval_piece_count
  - number of kittens and cats
  - cats are worth more
  - simple, but not very useful
- eval_board_bonus
  - pieces on board are more valuable than in hand
- eval_territory
  - center area more valuable
  - 2-in-a-row bonus
  - 7 on board bonus
  - L-shapes bonus
- eval_stranding
  - extra penalty if cats isolated on board, with no more in hand

```
def eval_territory(id: PlayerID, state: GameState):
  p1_score = _get_territory_score(PlayerID.ONE, state)
  p2_score = _get_territory_score(PlayerID.TWO, state)

  # apply anti-aggression measure
  if id == PlayerID.TWO:
      p1_score *= ANTI_AGGRESSION
  elif id == PlayerID.ONE:
      p2_score *= ANTI_AGGRESSION

  ply_penalty = _get_ply_penalty(id, state)
  win_bonus =_get_win_bonus(state)
  if win_bonus != 0:
        return win_bonus + ply_penalty

  return p1_score - p2_score + ply_penalty
```

Tuning Parameters

```
WIN_BONUS = 1000.0
PLY_PENALTY = 0.1
ANTI_AGGRESSION = 0.8

# a cat must be worth more than 3x kittens
CAT_MULTIPLIER = 25.0
KITTEN_MULTIPLIER = 1.0
BOARD_MULTIPLIER = 2.0
CENTER_MULTIPLIER = 2.5

PENDING_TRIPLE_BONUS = 4.0     # encourage getting triples
PENDING_PROMOTION_BONUS = 3.5  # encourage getting all pieces onto board
STRANDED_CAT_PENALTY = 1.0
```

Results

Evaluation Functions - *conducted with AlphaBetaAgent, maxStates 500,000 and depth 2*

1. eval_piece_count vs **eval_board_bonus** - *56 plies*
2. **eval_board_bonus** vs eval_territory - *55 plies*
3. **eval_board_bonus** vs eval_stranding - *55 plies*
4. **eval_territory** vs eval_board_bonus - *77 plies*

Beam Parameters - *eval_territory, ANTI_AGGRESSION changes to prevent deadlock*

1. **10 width, 10 depth** vs 5 width, 25 depth - *47 plies, ANTI_AGG = 0.5*
2. **20 width, 5 depth** vs 10 width, 25 depth - *53 plies, ANTI_AGG = 0.3*
3. **10 width, 25 depth** vs 20 width, 5 depth - *49 plies, ANTI_AGG = 0.5*

Alpha-Beta vs Beam Search - *eval_territory used for both*

1. Beam Search 10 width, 25 depth vs **AlphaBeta depth 2** - *22 plies*
2. **AlphaBeta depth 2** vs Beam Search 10 width, 25 depth - *27 plies*

Demonstration

Lessons Learned

- Evaluation functions are hard to get right!
  - Sometimes a small adjustment to one parameter makes a huge difference in behavior
  - AI can get stuck in cycles and deadlocks, especially vs itself
  - AI doesn't always go for the win, as it wants higher utility
- Rare cases in rules can take the longest to code
  - The "decisions" add complexity to minimax
  - This is especially true if players must act out of regular turn flow
- Visualization is a key for debugging
- Unit test the game logic - saves a lot of time and bugs in the long run

Future Work

- Better evaluation functions
- Optimization: can we exploit grid properties to save computations?
  - Example: equivalent moves based on grid rotation
- Reinforcement Learning Agent
  - Extract features from the Tuning Parameters and let the model train the weights
- Command line options - include in UI

Q/A