Marcel Kubík

Continental Rummy AI agents with human-like play

A CLI game engine with AI players that play imperfect games, with tunable parameters **memory**, **prediction** and **focus**. Each agent's capabilities have their cost. The agents play against each other in a tournament, achieving scores from their matches with other players and they are ranked based on their efficiency (average score / cost), not on the raw score. The game engine also supports interactive mode, where human player is able to play against any number of AI agents.

Algorithm and theory

The game

Continental Rummy is played with two 54-card decks (104 ranked cards + 4 jokers). Each player is dealt 14 cards. On a turn the player draws one card, optionally lays down melds and extends melds already present on the table, then discards (trashes) one card. A meld is either a set (3-4 cards of the same rank with distinct suits) or a run (at least 3 consecutive cards of one suit). Jokers can replace any card in the game except for a players one clean run within their melds. Laying melds on the table is not permitted until 3 rounds have passed, and a player's first meld must be worth at least 51 points. The first player to clean their hand of all cards wins the game. Opponents are penalised by the value left in their hands.

The core problem

The strategy in Continental Rummy is to minimize deadwood cards on hand (leftover cards which are not laid down to table in melds). This is a weighted set partitioning problem of choosing a collection of disjoint valid melds which cover as many cards as possible. In general, it is an NP-hard problem, however feasibly solvable for up to 15-card hands, so here it is solved exactly.

The solver (`best_decomposition`) uses memoised backtracking with a canonical sub-problem, always trying to cover the lowest uncovered natural card first. Such card is either deadwood, or it is the bottom of exactly one meld, so we every set is enumerated and every run that can contain it (with potentially jokers filling gaps), is recursed on the remainder, and the partition with the least deadwood is kept. This tends to use fewer jokers, so they are preserved.

The function (`hand_strength`) reduces a hand into a single number used for comparisons, e.g.: which card to discard. The function rewards melded cards and held jokers, and partially valuing weaker structures (pairs and same-suit adjacencies) which would potentially not be left as deadwood.

Human-like disadvantages

Human players do not play perfect games. They forget which cards they have seen, misread opponents' strategies and make occassional mistakes. Each disadvantage is modelled as an inverse of a capability.

- focus $\in [0,1]$ - with probability `focus` the agent chooses the best computed action, otherwise it makes a mistake and chooses another action randomly.
- memory $\in N$ - an agent only reads the last `memory x players` actions, and each remembered action survives with probability `exp(-age / memory)`. Low-memory agents

forget what an opponent discarded and their card-counting mechanism lacks complete information.

- prediction $\in N$ - controls how much the agent tries to infer opponents' intent. At 0 it ignores opponents and uses a flat prior for the stock. Higher values infer the neighoring cards of a card an opponent took from the trash that they are probably collecting a run. They also consider not feeding the opponents with cards they might find useful, when discarding.

These disadvantages are combined into card picking: `keep_value(hand - c)` $-\beta*$ `opponent_threat(c)`, where `keep_value` comes from `hand_strength`, `opponent_threat` from the memory-bounded opponent model, and β grows with `prediction`.

Cost and efficiency

Each capability is expensive, and the tournament's objective is efficiency (performance / cost)

```
cost = 1 + 0.40 * memory + 0.80 * prediction + 0.50 * (focus / (1 - focus))
```

Memory and prediction are linear, focus hyperbolic.

Implementation

The game is implemented purely in Python with standard modules. Cards are plain strings: `rank + suit`, e.g. `'10h'`, `'as'`, and `'jk'` for a joker. A hand is represented as a `dict[str, int]` (card: count), shared by reference with the owning `Player` class so the `AI` class always sees the current hand. The AI reads game state through a small interface (`game.actions`, `game.table`, `game.trash_top`, `game.rounds_elapsed`, `players[].hand_card_count`), so the decision-making is fully decoupled from the engine. The engine enforces game rules by raising exceptions whenever it encounters a violation.

A turn (action) is planned as an ordered list of steps the engine executes:

```
{'kind': 'meld',    'type': 'run', 'cards': [...]}   # lay a new meld
{'kind': 'layoff',  'card': c, 'table_idx': i}       # extend a table meld
{'kind': 'discard', 'card': c}                       # end the turn
{'kind': 'goout'}                                    # hand empty -> win
```

classes

- `Game` class owns the card stock, the discard pile, the shared table of melds, the `rounds_elapsed` counter, the action log.
- `Player` class holds the player's hand and either a human prompt loop or an `AI`. `turn()` method executes the AI's plan and independently validates the game rules before changing any state.
- `AI` class sees the pleyer's state and holds the perception of other player's hands and intentions

AI decision functions

- `observe` composes and opponent perception model about what each opponent seems to collect and a model about already seen cards, which are counted. The function reads only last `memory x players` actions, forgetting older ones.

- `threat` computes a threat score for discard candidate, a potential of helping an opponent
- `expected_deck_gain` is a Monte-Carlo estimation of a deck draw's value, sampling the distribution of remaining cards
- `decide_draw_action` compares gains of taking the top of discarded/trashed cards against taking from the deck
- `plan_turn` produces an ordered turn plan
- `decide_discard` picks the card to be discarded, maximizing `keep_value` $-\beta*$ `opponent_threat`. β rises with agent's prediction

Agent's capabilities (disadvantage) mechanism functions

- `apply_focus` manages taking the optimal solution with probability `focus` or taking an alternative
- `recall_probability` calculates probability of recalling an action from history. Probability of recall: `exp(-age/memory)`

Meld formation functions

- `melds_covering_first` considers every legal set and run that can contain the lowest uncovered card
- `solve`, `best_decomposition` partitioning the set of cards on hand into potential melds
- `hand_strength` reduces a hand to a single scalar number. Used when valuing and comparing potential draws and discards.
- `meld_value`, `melds_total_value` are useful in deciding whether the meld passes the threshold of 51 for the player's first meld.
- `can_extend_table_meld` decides whether a card can extend an already existing meld on the table

Installation and startup

Requirements

- Python 3.10

Running

```
python3 main.py
```

or, for an interactive game:

```
python3 main.py interactive
```

Running examples

Tournament

Multiple agents are generated, with distinct capabilities (disadvantages), playing against each other in the tournament. Upon winning a match, the winner scores 1 point, other players are penalized fractions of a point, based on their relative performance among the remaining players, possibly resulting in negative score. Implemented in `match_payoff` function.

Sample tournament output

```
running tournament of 27 agents, 702 games

rank    memory  prediction      focus   cost    average_score   score/cost
-------------------------------------------------------------------------------
1       1               0       0.7     2.37            0.57            0.24
2       4               0       0.7     2.97            0.68            0.23
3       1               2       0.7     3.37            0.70            0.21
4       8               2       0.7     4.77            0.76            0.16
5       8               0       0.7     3.77            0.57            0.15
6       1               0       0.9     5.70            0.83            0.15
7       4               0       0.9     6.30            0.92            0.15
8       4               2       0.7     3.97            0.57            0.14
9       8               0       0.9     7.10            1.02            0.14
10      4               4       0.7     4.97            0.67            0.14
11      1               4       0.7     4.37            0.53            0.12
12      4               2       0.9     7.30            0.88            0.12
13      1               2       0.9     6.70            0.79            0.12
14      4               4       0.9     8.30            0.94            0.11
15      1               0       0.3     1.41            0.16            0.11
16      8               4       0.9     9.10            0.96            0.11
17      8               2       0.9     8.10            0.78            0.10
18      8               4       0.7     5.77            0.51            0.09
19      1               4       0.9     7.70            0.64            0.08
20      1               2       0.3     2.41            0.18            0.08
21      4               0       0.3     2.01            0.04            0.02
22      8               2       0.3     3.81            0.05            0.01
23      1               4       0.3     3.41            0.04            0.01
24      4               4       0.3     4.01           -0.01           -0.00
25      8               4       0.3     4.81           -0.03           -0.01
26      4               2       0.3     3.01           -0.11           -0.04
27      8               0       0.3     2.81           -0.13           -0.05

MOST EFFICIENT AI AGENT: memory: 1 prediction: 0 focus: 0.7 (efficiency: 0.24)
```

Evaluation

The tournament ranked 27 agents (3 memory × 3 prediction × 3 focus levels) over 702 games by efficiency, defined as average score divided by capability cost. Performance was determined almost entirely by focus. Mean average score by focus level was 0.02 at 0.3, 0.62 at 0.7, and 0.86 at 0.9, with negative scores occurring only at focus 0.3. The highest `average_score` values correspond to highest focus (0.9), but the cost lowers the efficiency (score/cost). The winner

has the cheapest memory and prediction and 0.7 focus, therefore 70% of its decisions were made in favor of its best calculated actions.

Because the cost for focus rises hyperbolically, the best score/cost agent is a cheap, ~70% accurate, low-foresight player. Memory and prediction are barely feasible.

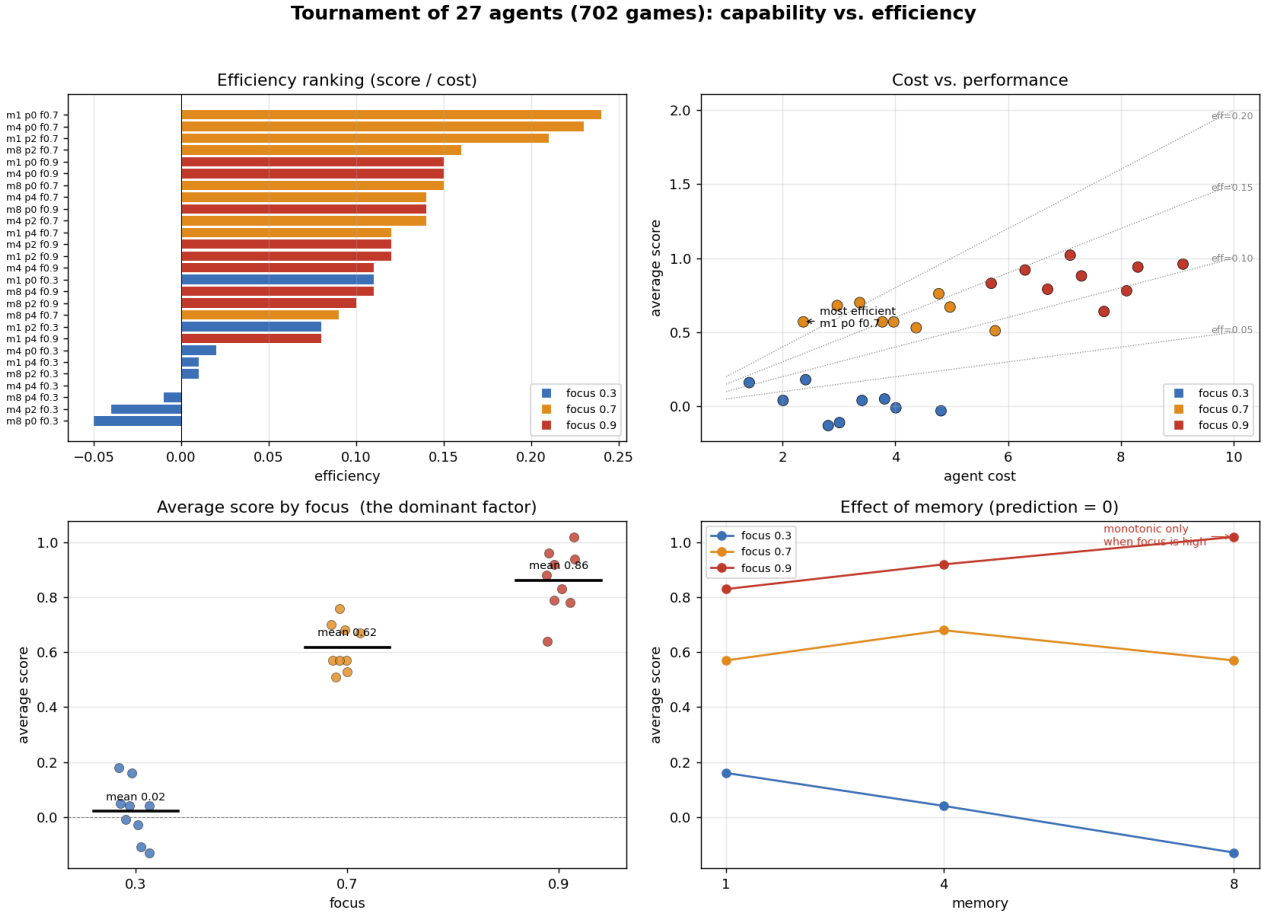

Figure 1: tournament of 27 agents

Related work

- Gin-Rummy:
  - https://github.com/ajyang99/Gin-Rummy
  - inspired `best_decomposition` function
- RLCard
  - https://rlcard.org/
  - provides reinforcement learning for card games (including Gin Rummy). However, goal of my project is the opposite of an optimal learned play.
- Bounded rationality: is a way of modelling sub-optimal human play; Represented by `focus` in this program. memory-bounded "what are they collecting?" perception layer.