Renju RDD

RDDについて

GitHub の方に、目次や項目内の記述に適切な定義に飛べるサイト内リンクを張っているMarkDownファイルをおいています。 実際に実装する場合は、そちらを参考にしてください。
(https://github.com/nagataaaas/renju/blob/main/RDD/RDD.md)

目次

        - init(moves: Array[Move])
        - compute()
        - compute_move(move: [Move])
        - check_foul(move: [Move])
        - is_win(program_number: int)
        - get_lines(program_number: int)
        - choose_next_move(program_number: int, depth: int = 1, best = 5)
        - find_options(line: Line)
        - line_extend_first(line: Line, foul_check: bool)
        - line_extend_second(line: Line, foul_check: bool)
        - available_extended_points(program_number: int, distance: int)
        - me
        - opponent(program_number: int)
        - copy()
        - is_black(program_number: int)
- Function
    - load_data(filename: str) -> Tuple[int, Array[Move]]
    - write_data(filename: str, table: Table)

Config

TABLE_SIZE

盤面の縦横の各大きさ。おそらく15でよい

TABLE_STARTS_WITH_ONE

盤面のデータをファイルから読み込む・ファイルに書き込む際、1-indexedにするかどうか。 授業内では明言されていなかった気がするので、一応実装しておく

RAISE_ON_CSV_COUNT_ERROR

ファイルから読み込んだデータに 総手数を表す数と、実際の手の数が異なる もしくは 手が交互に打たれていない という問題があったとき、 エラー(または勝利)にするかどうか。デバッグ時に適当に試したくなると思うので、その時用にどうぞ。

Enum

OptionType

> 1つの `Line` と最大その両側2つずつを先読みし、 `Option` の現状を示す値
>
> 優先順位は `Win > Checkmate > ToCheckmate > Winnable > Preferable > Trash`

- `Win` : すでに勝利条件を満たしている
- `Checkmate` : 対戦相手は勝利をとめることができない。達四や禁じ手によるもの
- `ToCheckmate` : "今"対戦相手が止めなければ、WinかCheckmateを作ってしまう
- `Winnable` : 対戦相手が止めなければ、勝てる
- `Preferable` : 勝ちに近づける。三を作れるなど。よしなに。
- `Trash` : あまり意味のない手。もしくは、打つことができない。

Direction

- `Horizontal` : 水平方向。右が正方向
- `Vertical` : 垂直方向。下が正方向
- `Diagonal_UL_BR` : 斜め方向。右下が正方向
- `Diagonal_UR_BL` : 斜め方向。左下が正方向

Class

Point

> 一点を表す。

init(y: int, x: int)

- `y` : y座標
- `x` : x座標

Line

> 一並びの連を表す。

init(direction: `Direction` , first: `Point` , second: `Point` , program_number: int)

- `direction` : `Line` の向き
- `first` : `Line` を作る両端の2点のうち、より負方向に近い方。長さ1(ただ一つの点で構成される点)の場合、 `first` と `second` は同じ。

- `second` : `Line` を作る両端の2点のうち、より正方向に近い方
- `program_number` : `Line` を作成したPlayerの番号

extend_first() -> Tuple[`Line` , `Point` , True] or Tuple[Null, Null, False]

自身を、負方向に長さ1つ伸ばすよう試みる。盤面内に収まれば、(伸ばされた後のLine, 新たに設置された `Point` , True), 不可能であれば `(Null, Null, False)` を返す。 盤面情報も加味した可否判断はTable.line_extend_firstに任せるので、ただ盤面範囲におさまるかの確認でかまわない。

extend_second() -> Tuple[`Line` , `Point` , True] or Tuple[Null, Null, False]

自身を、正方向に長さ1つ伸ばすよう試みる。盤面内に収まれば、(伸ばされた後の `Line` , 新たに設置された `Point` , True), 不可能であれば `(Null, Null, False)` を返す。 盤面情報も加味した可否判断は `Table.line_extend_second` に任せるので、ただ盤面範囲におさまるかの確認でかまわない。

length -> int

自身の長さを返却する `property`

Move

> "手"そのものを表す。

init(program_number: int, point: `Point`)

- `program_number` : 設置したPlayerの番号
- `point` : 設置した場所

Option

> その盤面において、指すことのできる場所、それ含め最短何ターンで勝利できるか、現在の状況を表す

init(type: `OptionType` , win_to: int, point: `Point`)

- `type` : 現状を表す
- `win_to` : この手を含め、あと何ターンで勝利できるか
- `point` : 該当箇所

add(option: `Option`) -> `OptionContainer`

> 自身と、 `option` を含めた `OptionContainer` を返す

- `option` : 一緒に `OptionContainer` に含める `Option`

OptionContainer

> 複数の `Option` をひとまとめにし、まとめた際の評価などを行う

init(options... : `Option...`)

- `options` : `Option` の可変長引数

add(option : `Option` or `OptionContainer`) -> `OptionContainer`

> 自身にoptionを追加、または拡張した `OptionContainer` を返す

- `option` : `Option` または `OptionContainer`

max -> `Option`

> 自身に格納されている `Option` の中で、最も `Option.type` の `優先度` が高いもの、特に、その中で最も `win_to` が 小さいものを返すproperty

winnable_with_skip -> bool

> 相手のターンを迎えた後、必ず勝てるかどうかのproperty
>
> `Win` や `Checkmate` の `Option` が格納されてるか, `ToCheckmate` の `Option` の `Option.point` が重複なしで2つ以上ならTrue, それ以外ならFalse

score -> int

> 自身を評価して数値化する。良しなにしてほしいが、 `winnable_with_skip` がTrueなら適当に大きな数字を返す、 とかは前提として考えてほしい

Table

> ゲームの盤面を表す

init(moves: Array[`Move`])

- `moves` : 盤面を構成する手の配列
- `moves_count` : `moves` の要素数
- `table` : keyを `Point` , valueをintとする連想配列。盤面の場所とそこにおいているPlayerを意味する。 keyとして存在しない `Point` は、まだ設置されていないことを意味する。設置されている点を全走査する際、 二次元配列は必ず盤面すべてを見なければならないが、連想配列は設置され

ている個数の走査で事足りる。 インスタンスの初期化段階では、この連想配列はメモリ確保のみで、 `moves` を用いたアップデートは行わない。

compute() -> Tuple[Null, Null] or Tuple[`Move` , bool]

> `moves` 内の全てのMoveを評価し、どちらかが五を作れば Tuple[その `Move` , True], どちらかが禁じ手を打つか既に打たれた 場所に設置すれば Tuple[その `Move` , False], どちらも無ければ Tuple[Null, Null] を返す

compute_move(move: [`Move`]) -> bool

> 与えられた `Move` を `check_foul` を用いて評価し、禁じ手かどうかを返却する。 禁じ手でなければ、自身の `moves` に追加し、 `moves_count` に1加算する。

- `move` : 評価する `Move`

check_foul(move: [`Move`]) -> bool

> 与えられた `Move` を評価し、禁じ手かどうかを返却する。

- `move` : 評価する `Move`

is_win(program_number: int) -> bool

> 該当の `program_number` があらわすPlayerが五を作ったかどうか

- `program_number` : 計算するPlayerの番号

get_lines(program_number: int) -> Map[int: Array[`Line`]]

> 該当の `program_number` があらわすPlayerが作成した全ての連を、その長さごとに配列に分け、長さをkeyにした連想配列に格納して返す。 `table` から、valueが `program_number` と等しいkeyの配列を作成し、各 `Direction` ごとに "配列から適当に `Point` を選び、length=0の `Line` を作る" -> "配列内にその `Point` が存在する場合に限り、 `Line` をその点のある方に伸ばす" を配列から要素が無くなるまで繰り返すことで、全ての方向の全ての `Line` を取得できる。

- `program_number` : 計算するPlayerの番号

choose_next_move(program_number: int, depth: int = 1, best = 5) -> Tuple[`Move` , `OptionContainer`]

> `program_number` にとって最良と思われる次の自身の `Move` を計算し、 `OptionContainer` とともに返す。 depthが1になるまで、次の相手の手を読まない段階で最良と思われる上位 `best` 個について `choose_next_move(対戦相手のnumber, depth-1, best)` を用いて相手の次の行動を予測し再帰的に評価する。

- `program_number` : 計算するPlayerの番号
- `depth` : この深さも含め、何回再帰的に計算するか。各再帰で1ずつ減算する。
- `best` : 上位何個について再帰的に計算するか

find_options(line: `Line`) -> `OptionContainer`

> 与えられた `Line` に対して、取り得る `Option` をまとめた `OptionContainer` を返却する。 `Line.length` が5以上の場合は `Line` 自体の評価を, 4の場合は両端を見た評価を、3, 2の場合は両端から2つ離れた `Point` までを見た評価を かえす。1の場合はよしなに(多分、そこまで細かくは見なくても強さはそんなに変わらないと思う)

- `line` : 評価する `Line`

line_extend_first(line: `Line` , foul_check: bool) -> Tuple[`Line` , `Point` , True] or Tuple[Null, Null, False]

> `Line.extend_first` と同様に負方向に `Line` を延長する。ただし、 `Line.extend_first` の 不可能判定に加え、新たに設置する `Point` が禁じ手になる場合も失敗とする。

line_extend_second(line: `Line` , foul_check: bool) -> Tuple[`Line` , `Point` , True] or Tuple[Null, Null, False]

> `Line.extend_second` と同様に負方向に `Line` を延長する。ただし、 `Line.extend_second` の 不可能判定に加え、新たに設置する `Point` が禁じ手になる場合も失敗とする。

available_extended_points(program_number: int, distance: int) -> Array[Point]

> 現状の `table` から、 `get_lines(program_number)` を用いて全ての `Line` を取り出し、それぞれを両側に対して最大 `distance` まで伸ばして得られる新たな `Point` について、 禁じ手で無いものの配列を返す

- `program_number` : 計算するPlayerの番号
- `distance` : 両側の距離どれだけまで返却するか

me -> int

> 次に手を打つPlayerの番号を返却するproperty。 `moves_count` が1以下の場合は、適当に作成する

opponent(program_number: int) -> int

> 与えられた `program_number` の対戦相手にあたるPlayerの番号を返却する

- `program_number` : 計算するPlayerの番号

copy() -> `Table`

> 自身のコピーを返却する

is_black(program_number: int) -> bool

> 与えられた `program_number` が先手の番号かどうかを返却する。 `moves_count` が0の時、`program_number` にかかわらずTrueを返す

- `program_number` : 計算するPlayerの番号

Function

load_data(filename: str) -> Tuple[int, Array[`Move`]]

> 与えられた `filename` の表すファイルからデータを読み込む。 この時、`Config.TABLE_STARTS_WITH_ONE` と `Config.RAISE_ON_CSV_COUNT_ERROR` を使うこと。 手が書き込まれていないこと(空ファイル, `0`, など)や、そもそもファイルが存在しない事も考えること。

- `filename` : 読み込み元のファイル名

write_data(filename: str, table: `Table`)

> 与えられた `filename` の表すファイルにデータを書き込む。 この時、`Config.TABLE_STARTS_WITH_ONE` を使うこと。 `table.moves_count` が0のとき(書き込む意味はないけど)は `0`, を出力する。

- `filename` : 書き込み先のファイル名