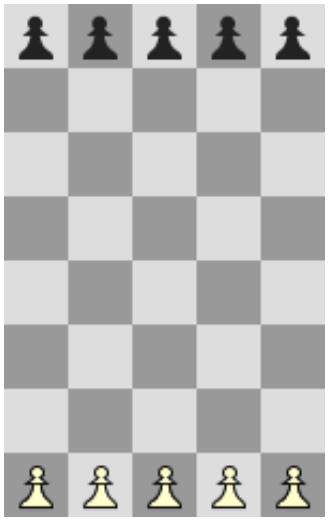
Jogo dos Peões

1 O Jogo

1.1 Descrição (versão desafio)

Usando apenas 5 colunas de um tabuleiro de Xadrez, 5 peões pretos e 5 peões brancos, dois jogadores movimentam esses peões alternadamente até que um deles consiga colocar os seus peões nas casas inicialmente ocupadas pelos peões do adversário. Eis a posição inicial do tabuleiro:



Posição inicial.

Cada jogador, na sua vez, pode mover apenas um peão e este deve permanecer na mesma coluna. Só são permitidas jogadas em que o peão movimentado avança em direção ao campo do adversário. Um peão pode avançar um número arbitrário de casas livres. Se, no momento da jogada, ele estiver sendo bloqueado pelo peão do adversário, então ele pode "pular", mas deve parar na casa livre logo em seguida.

Pode acontecer que um jogador fique sem movimentos válidos na sua vez de jogar. Nesse caso, ele é obrigado a "passar" a vez para o adversário.

1.2 Tarefa

Sua tarefa é fazer um *bot* capaz de jogar este jogo, ou seja, você deve fazer um jogador virtual que seja capaz de ganhar o jogo dos peões sempre que existir uma estratégia vencedora.

1.3 Conceitos abordados

A resolução deste problema envolve conhecimentos de combinatória, teoria dos grafos, estrutura de dados, recursão, simetrias, e o conceito de pré-processamento.

1.3.1 Combinatória

Em primeiro lugar, para estimar o número de estados desse jogo, é necessário conhecer o princípio fundamental da contagem.

1.3.2 Teoria de grafos

De cada uma das possíveis configurações do jogo, e dependendo de quem é a vez de jogar, é possível efetuar um conjunto de movimentos válidos que pode ser vazio. Portanto, é natural modelar o jogo como um grafo dirido, onde os vértices do grafo são todas as possíveis configurações do tabuleiro e os arcos desse grafo possuem duas cores: os arcos brancos representam os movimentos válidos para o jogador dos peões brancos e os arcos pretos têm função análoga.

1.3.3 Estruturas de dados, simetrias, pré-processamentos

O problema de se encontrar uma estratégia vencedora não é trivial por causa do tamanho do espaço de estados e, portanto, não se pode representar o grafo todo na memória RAM. O problema se agrava, por exemplo, se o jogo estiver sendo implementado para rodar em dispositivos móveis (e.g. celular). Uma solução para esse problema é explorar as simetrias do jogo para encontrar uma estrutura de dados, capaz de representar esse grafo, e que seja econômica do ponto de vista de memória.

A eficiência do cálculo da estratégia ótima pode ser aumentada com o uso de tabelas pré-computadas com os valores de funções que são muito usadas.

1.3.4 Recursão

O algoritmo que encontra a estratégia vencedora (e o jogador que a possui) é essencialmente uma busca em profundidade e, portanto, possui natureza recursiva.

2 Versão simplificada

Para facilitar o entendimento da solução do problema, vamos detalhar a solução de uma versão simplificada do problema, usando apenas 3 colunas do tabuleiro e supondo que o número de linhas é 5. Veja a figura abaixo com a posição inicial do jogo simplificado.

Vamos chamar os jogadores de 0 e 1 e vamos supor que o jogador 0 controla os peões brancos e que o jogador 1 controla os peões pretos.

2.1 Configurações e estados

Vamos usar a palavra *configuração* para denotar a posição das peças no tabuleiro (ou em uma coluna). Vamos usar a palavra estado para denotar, além da configuração, a vez da jogada. Portanto, para cada configuração, existem dois estados possíveis: um em que a vez da jogada é do jogador 0 e outro em que a vez da jogada é do jogador 1.

Versão simplificada

2.1.1 Contando configurações e estados

Em cada coluna, o peão preto pode estar em qualquer das 5 casas. Uma vez que o peão preto ocupa uma dessas casas, o peão branco pode estar em uma das 4 casas restantes. Portanto são 20 configurações possíveis para cada coluna. Como o tabuleiro possui 3 colunas, são 20^3 possíveis configurações do tabuleiro no total.

Além dessas 8000 configurações do tabuleiro, pode ser a vez dos peões brancos se moverem ou pode ser a vez dos peões pretos se moverem. Portanto o jogo possui 16000 estados. *Na verdade, nem todos esses estados são atingíveis durante um jogo. Você consegue dar exemplos de estados inatingíveis?*

```
In [ ]: # número de estados do jogo:
        2 * 20**3
```

2.1.2 Configurações de uma única coluna

Nesta seção, vamos enumerar as 20 possíveis configurações de uma coluna. Cada uma dessas configurações receberá um identificador único que é um inteiro variando de 0 a 19. Veja as funções abaixo e entenda o que elas fazem e por que elas funcionam.

```
In [3]: # p é o índice da casa ocupada pelo peão preto
        # b é o índice da casa ocupada pelo peão branco
        def imprime_coluna(b, p):
            for i in range(5):
                if p == i:
                    print("P", end="")
                elif b == i:
                    print("B", end="")
                else:
                    print(".", end="")

        def enumera_configs_coluna():
```

```
            id = 0
            for b in range(5):
                for p in range(5):
                    if b != p:
                        imprime_coluna(b, p)
                        print(":", id)
                        id += 1

        enumera_configs_coluna()
```

```
BP...: 0
B.P..: 1
B..P.: 2
B...P: 3
PB...: 4
.BP..: 5
.B.P.: 6
.B..P: 7
P.B..: 8
.PB..: 9
..BP.: 10
..B.P: 11
P..B.: 12
.P.B.: 13
..PB.: 14
...BP: 15
P...B: 16
.P..B: 17
..P.B: 18
...PB: 19
```

Durante a execução do programa vamos precisar computar o seguinte. Dado um índice $0 \leq id < 20$ de uma configuração, precisaremos saber qual a posição do peão preto (e também do branco) na configuração *id*. Como vamos precisar disso muitas vezes, podemos pré-computar essas posições no início do programa e guardá-las na memória, nos vetores `pos[0]` e `pos[1]`.

- `pos[0][id]` é o índice da casa ocupada pelo peão branco numa coluna de configuração *id*;
- `pos[1][id]` é o índice da casa ocupada pelo peão preto numa coluna de configuração *id*.

Veja como a estrutura da função `gerar_vetores_posicao_coluna()` é muito parecida com a estrutura da função `enumera_config_coluna()`.

```
In [4]: pos = [list(range(20)), list(range(20))]

        def gerar_vetores_posicao_coluna():
            id = 0
            for b in range(5):
                for p in range(5):
```

```
                if b != p:
                    pos[0][id] = b
                    pos[1][id] = p
                    id += 1

        gerar_vetores_posicao_coluna()
```

Para ver como ficaram preenchidos os vetores `pos[0]` e `pos[1]` podemos fazer:

```
In [5]: pos[0]

Out[5]: [0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4]

In [6]: pos[1]

Out[6]: [1, 2, 3, 4, 0, 2, 3, 4, 0, 1, 3, 4, 0, 1, 2, 4, 0, 1, 2, 3]
```

Os valores impressos batem com o que você estava esperando?

Veja que `pos[1][6] = 3` por que a configuração associada com o *id* 6 é `.B.P.` e, nessa configuração, o peão preto ocupa a posição de índice 3 (i.e. a 4ª posição da esquerda para a direita). Um argumento semelhante justifica por que `pos[0][6] = 1`.

Também será conveniente manter uma tabela `oid[b][p]` que guarda, para cada posição $0 \leq b < 5$ e $0 \leq p < 5$ o *id* da configuração que possui o peão preto na casa p e o peão branco na casa b.

```
In [7]: oid = [[-1 for i in range(5)] for j in range(5)]

        def preenche_oid():
            for id in range(20):
                oid[pos[0][id]][pos[1][id]] = id

        preenche_oid()
```

2.1.3 Configurações do tabuleiro (três colunas)

Até agora nos preocupamos apenas com uma coluna do tabuleiro, mas ele possui 3 colunas! Precisamos, portanto, dar um jeito de indexar todas as possíveis configurações do tabuleiro. Como cada coluna tem 20 configurações possíveis, serão $20 \times 20 \times 20 = 8000$ configurações no total. Precisamos associar a cada uma delas um *ID* de 0 a 7999.

Podemos usar "base 20" para representar as configurações. Por exemplo, se a primeira coluna estiver na confuguração `...BP` (*id* 15), a segunda coluna estiver em `.PB..` (*id* 9) e a terceira, em `B..P.` (*id* 2), então o *ID* dessa configuração do tabuleiro será

$$20^0 \times 15 + 20^1 \times 9 + 20^2 \times 2$$

o que resulta em 995 (em base 10).

```
In [8]: # conferindo:
        15 + 20 * 9 + 20**2 * 2
```

```
Out[8]: 995
```

É uma boa idéia criarmos funções de conversão que mapeiam 3 *ids* de colunas para um *ID* do tabuleiro e vice versa.

```
In [9]: # Recebe os ids das configurações das colunas 0, 1 e 2 e
        # calcula o ID correspondente da configuração do tabuleiro
        def get_ID(id0, id1, id2):
            return 400 * id2 + 20 * id1 + id0

        # Recebe o ID da confugração do tabuleiro e extrai o id da
        # coluna col (onde 0 <= col < 3)
        def get_col_id(ID, col):
            while col > 0:
                ID = ID // 20
                col = col - 1
            return ID % 20
```

Vamos pré-computar os valores da segunda função. Você saberia dizer se vale a pena pré-computar os valores da primeira função? Por que?

```
In [10]: # Pré-computando todos os possíveis valores da segunda função...
         col_id = [[get_col_id(ID, col) for col in range(3)] for ID in range(8000)]
```

Para quem não está acostumado com a sintaxe de Python, o código acima é um exemplo de *list comprehension* e equivale ao seguinte código:

```
In [11]: for ID in range(8000):
             for col in range(3):
                 col_id[ID][col] = get_col_id(ID, col)
```

Para usar esses valores pré-computados basta acessar `col_id[ID][col]`, que guarda exatamente o mesmo valor que o resultado da chamada `get_col_id(ID, col)`.

Um jeito um pouco mais rápido de preencher a tabela `col_id` é o seguinte:

```
In [12]: col_id = [[-1 for col in range(3)] for ID in range(8000)]

         for i in range(20):
             for j in range(20):
                 for k in range(20):
                     ID = get_ID(i, j, k) # mais rápido ainda: 400*k + 20*j + i
                     col_id[ID][0] = i
                     col_id[ID][1] = j
                     col_id[ID][2] = k
```

Porque não tabelamos a função `get_ID()`?

Em C, por exemplo, a expressão `M[i][j][k]` é traduzida para `*(*(*(M + i) + j + k)` o que envolve 3 somas e 3 derreferenciações. Em outras linguagens, o acesso a matrizes tridimensionais não é muito diferente. Portanto, acessar uma tabela tridimensional ou fazer as contas dentro da função `get_ID()` custa quase a mesma coisa (exceto o *overhead* de se chamar a função).

2.1.4 Configuração inicial

Vamos fixar `B...P` (*id* 3) como sendo a configuração inicial de uma coluna. Portanto, a configuração inicial do jogo é o ID retornado por `get_ID(3,3,3)`. Isso significa que os peões pretos só podem se mover para casas de índice menor do que o atual. De maneira simétrica, peões brancos só podem se mover para casas de índice maior.

```
In [13]: get_ID(3,3,3)

Out[13]: 1263
```

2.1.5 Configurações terminais (fim de jogo)

Quando olhamos para uma coluna individualmente, qualquer configuração de coluna pode ocorrer antes do jogo terminar. Contudo, quando olhamos para o tabuleiro todo, algumas configurações só ocorrem no fim do jogo: são aquelas em que os peões de uma determinada cor preenchem as casas originalmente ocupadas pelos peões do oponente.

A função a seguir identifica se uma configuração é temrinal e retorna informação suficiente para saber qual jogador completou o jogo.

```
In [14]: def terminal(ID):
             ganhou0 = 0
             ganhou1 = 0
             v = [col_id[ID][i] for i in range(3)]
             # jogador 0 já ganhou
             if pos[1][3] == pos[0][v[0]] and \
                pos[1][3] == pos[0][v[1]] and \
                pos[1][3] == pos[0][v[2]]:
                 ganhou0 = 1
             # jogador 1 já ganhou
             if pos[1][v[0]] == pos[0][3] and \
                pos[1][v[1]] == pos[0][3] and \
                pos[1][v[2]] == pos[0][3]:
                 ganhou1 = 1
             return ganhou0 + 2 * ganhou1
```

Note que a função `terminal()` devolve 0, 1, 2 ou 3 (em binário 00, 01, 10, 11). O primeiro bit (bit menos significativo) é 0 ou 1 dependendo do jogador 0 (peões brancos) ter atingido a posição desejada ou não. O segundo bit dá a informação correspondente ao jogador 1 (peões pretos).

Qual a configuração para a qual `terminal()` devolve 3?

2.1.6 Imprimindo uma configuração do tabuleiro

```
In [15]: def imprime_config_coluna(id):
             imprime_coluna(pos[0][id], pos[1][id])

         def imprime_config(ID):
             for i in range(3):
```

```
            id = col_id[ID][i]
            imprime_config_coluna(id)
            print()
        print()
```

2.2 O grafo de movimentos válidos

2.2.1 Movimentos válidos em 1 coluna

Na vez de cada jogador, ele escolhe uma coluna e move seu peão. Portanto, para cada possível configuração de uma coluna, e para cada jogador, existe uma lista de movimentos válidos. Vamos representar o grafo de movimentos válidos de cada jogador por um vetor de listas.

- `grafo_coluna[0][id]` é a lista de configurações que podem ser atingidas a partir da configuração *id* através de 1 movimento válido do peão branco.
- `grafo_coluna[1][id]` é a lista de configurações que podem ser atingidas a partir da configuração *id* através de 1 movimento válido do peão preto.

A função a seguir recebe o *id* de uma configuração de uma coluna, e calcula a lista dos *ids* das configurações que o jogador 0 (que controla os peões brancos) consegue atingir com um único movimento válido de seu peão.

```
In [16]: def movimentos_validos_coluna_0(id):
             b = pos[0][id]
             p = pos[1][id]
             if b == 4:
                 # o branco já chegou no final
                 return []
             if b == 3 and p == 4:
                 # aqui o branco está na penúltima casa, mas
                 # não pode se mover porque o preto está na última casa
                 return []
             if p == b + 1:
                 # o branco só pode "pular" o preto e parar
                 # na casa imediatamente após o preto
                 return [oid[b + 2][p]]

             # Se nenhuma das condições anteriores foi satisfeita, então o peão
             # branco pode se mover para qualquer casa entre ele até peão preto ou
             # até o fim da linha no caso dele já ter ultrapassado o peão preto.
             validos = []
             limite = 5 if p < b else p

             for i in range(b + 1, limite):
                 validos.append(oid[i][p])
             return validos

         grafo_coluna = [[], []]
         grafo_coluna[0] = [movimentos_validos_coluna_0(id) for id in range(20)]
```

Para calcular a lista dos movimentos válidos do peão preto, não é necessário reescrever a função. Basta fazermos uma função que inverte preto/branco e esquerda/direita, usar os movimentos válidos já calculados para o peão branco e inverter de volta em cada possível configuração da lista obtida.

```
In [17]: def inverte(id):
             b = pos[0][id]
             p = pos[1][id]
             return oid[4 - p][4 - b]
```

A conta `4 - x` serve como um espelho com relação a esquerda e direita. Por exemplo, ao chamarmos a função `inverte()` na configuração `B.P..` (*id* 1), o valor de retorno consiste na sequencia espelhada obtida após trocarmos P e B de lugar:

```
config original:    B.P..
 trocando P e B:    P.B..
     espelhando:    ..B.P
```

Portanto, `inverte(1) = 11`.

Estamos prontos para calcular as listas de movimentos válidos do peão preto. Veja abaixo:

```
In [18]: grafo_coluna[1] = [[inverte(j) for j in grafo_coluna[0][inverte(id)]] \
                            for id in range(20)]
```

Podemos verificar se, de fato, `grafo_coluna[0][id]` (resp. `grafo_coluna[1][id]`) guarda os movimentos válidos do peão branco (resp. preto) a partir da configuração *id*.

```
In [19]: print('Movimentos válidos do peão branco:')
         for id in range(20):
             imprime_config_coluna(id)
             print(": ", end="")
             for j in grafo_coluna[0][id]:
                 imprime_config_coluna(j)
                 print("  ", end="")
             print()
```

```
Movimentos válidos do peão branco:
BP...: .PB..
B.P..: .BP..
B..P.: .B.P.  ..BP.
B...P: .B..P  ..B.P  ...BP
PB...: P.B..  P..B.  P...B
.BP..: ..PB.
.B.P.: ..BP.
.B..P: ..B.P  ...BP
P.B..: P..B.  P...B
.PB..: .P.B.  .P..B
..BP.: ...PB
..B.P: ...BP
```

```
P..B.: P...B
.P.B.: .P..B
..PB.: ..P.B
...BP:
P...B:
.P..B:
..P.B:
...PB:
```

```
In [20]: print('Movimentos válidos do peão preto:')
         for id in range(20):
             imprime_config_coluna(id)
             print(": ", end="")
             for j in grafo_coluna[1][id]:
                 imprime_config_coluna(j)
                 print("  ", end="")
             print()
```

```
Movimentos válidos do peão preto:
BP...:
B.P..: BP...
B..P.: B.P..  BP...
B...P: B..P.  B.P..  BP...
PB...:
.BP..: PB...
.B.P.: .BP..
.B..P: .B.P.  .BP..
P.B..:
.PB..: P.B..
..BP.: .PB..
..B.P: ..BP.
P..B.:
.P.B.: P..B.
..PB.: .P.B.  P..B.
...BP: ..PB.
P...B:
.P..B: P...B
..P.B: .P..B  P...B
...PB: ..P.B  .P..B  P...B
```

2.2.2 O grafo de movimentos válidos (3 colunas)

Baseando-nos no grafo de movimentos em uma única coluna, somos capazes de gerar o grafo de movimentos válidos do jogo todo. Suponha que $\vec{G} = (V, E)$ seja o grafo dirigido dos movimentos válidos para um determinado jogador em uma coluna. O grafo de movimentos válidos para o mesmo jogador no tabuleiro todo é isomorfo ao grafo $\vec{H} = (V_H, E_H)$, definido a seguir. Primeiramente, definimos V_H como

$$V_H = V^3.$$

Portanto, um vértice $v \in V_H$ possui as coordenadas v_1, v_2 e v_3 que correspondem às configurações de cada coluna. Esse vértice corresponde à uma configuração do tabuleiro todo (3 colunas). Depois, definimos $T \subseteq V_H$ como sendo o conjunto de estados terminais do jogo. Em seguida, definimos o conjunto de arcos desse grafo dirigido como

$$E_H = \big\{(u,v) \in (V_H \setminus T) \times V_H \colon \exists\, i \in \{1,2,3\}\ \big((u_i, v_i) \in E \wedge (\forall\, j \in \{1,2,3\}, j \neq i \implies u_j = v_j)\big)\big\},$$

ou seja, uma configuração (u_1, u_2, u_3) pode ser levada à configuração (v_1, v_2, v_3) através de um movimento válido desse jogador se duas das coordenadas se mantiverem iguais e uma terceira for alterada de acordo com o grafo de movimentos válidos de uma coluna (i.e. de acordo com os arcos do grafo $\vec{G}$). Afinal, o jogador só pode mover um peão por vez. A condição $u \notin T$ traduz o fato de que nenhuma jogada pode ser feita depois que o jogo atingiu um estado terminal.

2.2.3 Representação do grafo na memória

Matriz de adjacência. Uma maneira ingênua de representar esse grafo seria alocar uma matriz M de dimensões 8000×8000, onde M_{ij} vale 1 se $i \to j$ é um movimento válido e 0 caso contrário.

Essa representação ocuparia 16Mb de memória (ou 2Mb se comprimíssemos a representação usando um bit por entrada da matriz). Mas, pior do que isso, seria o fato de que, para percorrer a lista de configurações acessíveis (por movimentos válidos) a partir de uma configuração i, precisaríamos percorrer a i-ésima linha da matriz que tem 8000 entradas: muito ruim, dado que o número máximo de vizinhos de um vértice (configurações atingíveis por 1 movimento válido) é 9.

Pelos motivos elencados acima, não é viável utilizar uma matriz de adjacência.

Listas de adjacência Um outro jeito de armazenar o grafo na memória é através do uso de listas de adjacências. Essa estrutura de dados consiste em armazenar, para cada vértice do grafo, uma lista dos seus vizinhos. Lambre-se de que já usamos esta estrutura de dado para representar um grafo, por exemplo, quando criamos o vetor `grafo_coluna[0]` que armazena o grafo de movimentos válidos do jogador 0 em uma única coluna.

Como são 8000 configurações indexadas de 0 a 7999, vamos precisar de pelo menos $\lceil \log_2 8000 \rceil$ bits para representar o ID de uma configuração.

```
In [21]: from math import log, ceil

         ceil(log(8000, 2))
```

```
Out[21]: 13
```

Arredondando esse número para um múltiplo de 8, precisamos de 16 bits (2 bytes) para representar uma configuração na memória.

Para cada uma dessas 8000 configurações, precisamos armazenar uma lista das configurações vizinhas nesse grafo. São 3 vizinhos, no máximo, para cada coluna (confira na a tabela que enumera as configurações de uma coluna). Portanto, são 9 estados vizinhos, no máximo, para cada uma das 8000 configurações do tabuleiro.

Podemos então armazenar um vetor com 8000 entradas, cada uma sendo uma lista com no máximo 9 elementos com 16 bits cada. Na j-ésima posição da i-ésima lista estaria armazenado o *ID* do j-ésimo vizinho da configuração com *ID* i.

Essa representação ocupa, no máximo, 144Kb conforme a conta abaixo.

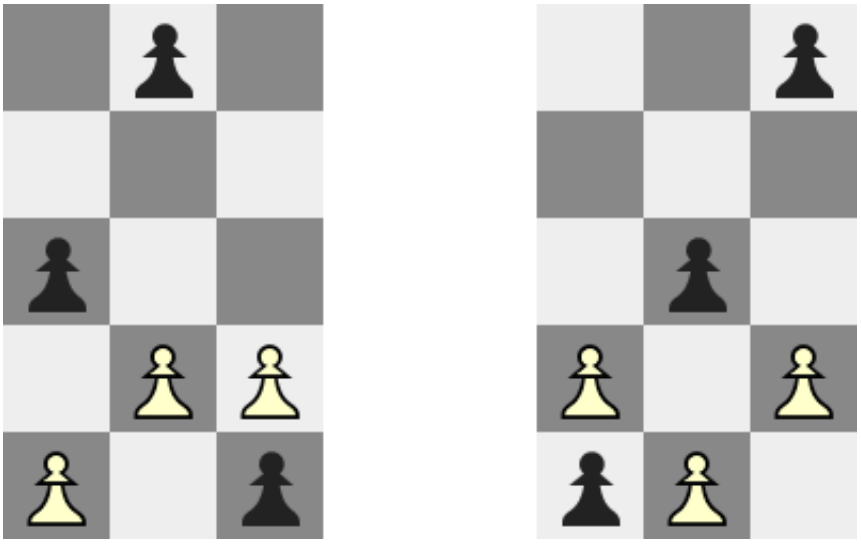
```
In [22]: # número de bytes aproximadamente:
         8000 * 9 * 2

Out[22]: 144000

In [23]: def movimentos_validos_tabuleiro(ID, jogador):
             if terminal(ID):
                 return []
             v = col_id[ID]
             resposta = []
             for i in range(3):
                 u = v.copy()
                 for id in grafo_coluna[jogador][v[i]]:
                     u[i] = id
                     x = get_ID(*u)
                     resposta.append(x)
             # Se não há jogadas válidas então "passar a vez" é válida!
             if len(resposta) == 0:
                 resposta.append(ID)
             return resposta

         grafo_tabuleiro = [[], []]

         for jogador in range(2):
             grafo_tabuleiro[jogador] = [movimentos_validos_tabuleiro(ID, jogador) \
                                         for ID in range(8000)]
```

Apesar dessa quantidade de memória ser bem inferior aos 16Mb que uma matriz de adjacência ocuparia, precisamos otimizar ainda mais o consumo de memória para que possamos implementar a solução do problema original (8 linhas $\times$ 5 colunas) de forma eficiente em dispositivos com pouca memória (e.g. celular). Para tanto, precisamos explorar as simetrias do grafo (e portanto as simetrias do jogo), e é isso que faremos na próxima seção.

2.3 Explorando simetrias

Observação: se você está implementando a primeira versão do código e se não pretende implementar a versão do exercício que utiliza o tabuleiro 5x8, você pode pular essa seção e ir direto para a seção Section 2.4

Note que o jogo, tanto na versão original como na versão simplificada, consiste de várias colunas que são independentes. Por esse motivo, a ordem das colunas não interfere na existência de uma estratégia vencedora para um determindado jogador. Por exemplo, se o jogador que controla os peões pretos tem uma estratégia vencedora para a configuração da esquerda, então ele também tem uma estratégia vencedora para a configuração da direita.



Tabuleiros equivalentes.

No exemplo da figura acima, vamos chamar a configuração da esquerda de v e a da direita de w. Temos que v consiste de três *ids* v_1, v_2 e v_3 (um para cada coluna). Do mesmo modo, $w = (w_1, w_2, w_3)$. Dizemos que v e w são configurações *equivalentes* pois w é uma permutação de v. Mais especificamente, $w = (v_3, v_1, v_2)$.

Portanto, se a partir de uma configuração $u \in V_H$ fosse possível ir para a configuração v, então poderíamos trocar v por w na lista dos vizinhos de u sem essencialmente mudar o jogo. Isto é, essa troca não traria vantagens ou desvantagens para nenhum dos jogadores.

É claro que essa mudança só pode ser feita internamente, na estrutura de dados que representa o grafo na memória, pois na interface gráfica com os usuários (jogadores), se um deles fizer o movimento $u \to v$, a configuração resultantee que devemos exibir aos usuários ainda deve ser v e não w, senão os jogadores ficariam confusos sobre o que está acontecendo no jogo.

2.3.1 O grafo quociente (ou grafo reduzido)

Nesta seção pretendemos dar os detalhes de como podemos usar o que foi dito anteriormente para economizar memória. Em primeiro lugar podemos dividir as configurações em "grupos" de configurações equivalentes e eleger um representante para cada grupo. Daí olhamos para o grafo projetado no conjunto de vértices representantes.

Exemplo de grupo Por exemplo, se olharmos para a configuração 995 que é representada pela tripla

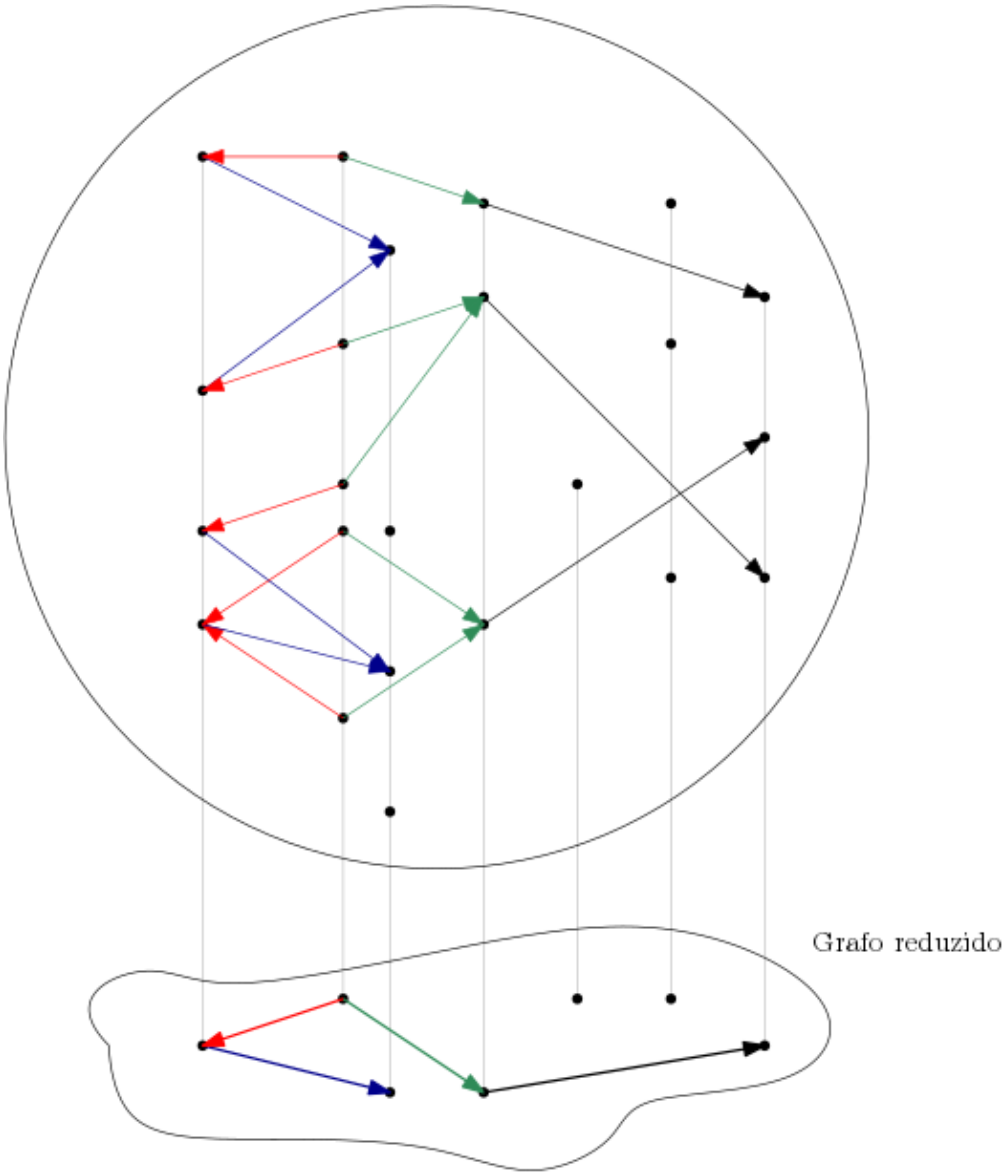
$$(15, 9, 2)$$

podemos listar facilmente as configurações no seu grupo:

```
 995 (15, 9, 2)
3655 (15, 2, 9)
6049 (9, 2, 15)
1109 (9, 15, 2)
```

```
3902 (2, 15, 9)
6182 (2, 9, 15)
```

Já a configuração $(3,3,3)$, que é a configuração inicial do jogo, está sozinha no seu grupo pois qualquer permutação desse vetor é identica ao próprio vetor.



Grafo reduzido

Elegendo o representante de cada grupo Devemos criar uma função Φ que associa cada vértice $v \in V_H$ ao representante do grupo que contém v. Um modo simples de fazer isso é tomar como representante a tripla dentro do grupo que está ordenada de maneira não-decrescente (ou, equivalentemente, a tripla lexicograficamente menor). Portanto, podemos definir $\Phi\colon V_H \to V_H$ de modo que $\Phi(v) = (v_{\pi(1)}, v_{\pi(2)}, v_{\pi(3)})$, onde $\pi\colon \{1,2,3\} \to \{1,2,3\}$ é uma permutação que satisfaz

$$v_{\pi(1)} \leq v_{\pi(2)} \leq v_{\pi(3)}.$$

Podemos codificar a função `phi()` em Python da segunite maneira:

```
In [ ]: def phi(ID):
            a = col_id[ID][0]
            b = col_id[ID][1]
            c = col_id[ID][2]
            if a <= b:
                if b <= c:
                    return ID
                if a <= c:
                    return get_ID(a, c, b)
                return get_ID(c, a, b)
            if a <= c:
                return get_ID(b, a, c)
            if b <= c:
                return get_ID(b, c, a)
            return get_ID(c, b, a)
```

Podemos fazer uma função semelhante para obter a permutação π:

```
In [ ]: def pi(ID):
            a0 = col_id[ID][0]
            a1 = col_id[ID][1]
            a2 = col_id[ID][2]
            rID = phi(ID)
            b0 = col_id[rID][0]
            b1 = col_id[rID][1]
            b2 = col_id[rID][2]
            if a0 == b0:
                if a1 == b1:
                    return (0, 1, 2)
                return (0, 2, 1)
            elif a0 == b1:
                if a1 == b0:
                    return (1, 0, 2)
                return (2, 0, 1)
            if a1 == b0:
                return (1, 2, 0)
            return (2, 1, 0)
```

Se precisarmos calcular a permutação inversa:

```
In [ ]: def inv(p):
            q = list(range(3))
            for i in range(3):
                q[p[i]] = i
            return tuple(q)
```

Grafo reduzido O grafo reduzido pode ser formalizado da seguinte forma. Seja $V_R = Img(\Phi)$ o conjunto imagem da função Φ, isto é $V_R \subseteq V_H$ é o conjunto de vértices que são representantes de algum grupo. O grafo reduzido é o grafo dirigido $\vec{R} = (V_R, E_R)$, onde o conjunto de arcos é definido por

$$E_R = \{(\Phi(u), \Phi(v)) : (u, v) \in E_H\}.$$

Estamos prontos para armazenar o grafo reduzido de movimentos válidos na memória.

```
In [ ]: # Lembre-se de que:
        #    * o jogador 0 controla os peões brancos,
        #    * o jogador 1 controla os peões pretos.

        representantes = {}

        for ID in range(8000):
            if terminal(ID) == 0:
                representantes[phi(ID)] = True

        # grafo_tabuleiro[i] -- armazena o grafo dos movimentos válidos do jogador i
        grafo_tabuleiro = [{}, {}]

        for ID in representantes.keys():
            v = [col_id[ID][i] for i in range(3)]
            for jogador in range(2):
                grafo_tabuleiro[jogador][ID] = []
                for i in range(3):
                    u = v.copy()
                    for id in grafo_coluna[jogador][v[i]]:
                        u[i] = id
                        x = phi(get_ID(*u))
                        if x not in grafo_tabuleiro[jogador][ID]:
                            grafo_tabuleiro[jogador][ID].append(x)
                # Verifica se é preciso permitir a jogada "passar a vez"
                if len(grafo_tabuleiro[jogador][ID]) == 0:
                    grafo_tabuleiro[jogador][ID].append(ID)
```

No trecho de código acima, a variável `representantes` é um dicionário. Em outras linguagens de programação, é possível implementar um dicionário usando uma árvore binária.

2.4 O grafo de estados do jogo

Agora que já sabemos como as configurações podem ser levadas umas às outras através de movimentos válidos dos peões pretos e brancos, vamos criar o grafo de estados que modela todos os aspectos do jogo: configuração do tabuleiro e vez da jogada.

Lembre-se: um estado nada mais é do que uma configuração junto com a informação de qual é o jogador da vez.

```
In [25]: grafo = {}
```

```
# i varia no conjunto de jogadores {0, 1}
for i in range(2):
    # Se você implementou o grafo reduzido descrito na seção anterior,
    # então use representantes.keys() no lugar de range(8000) a seguir
    for x in range(8000):
        grafo[(i, x)] = []
        for y in grafo_tabuleiro[i][x]:
            grafo[(i, x)].append((1 - i, y))
```

Para cada jogador i e configuração x, armazena-se em `grafo[(i,x)]` a lista de estados atingíveis a partir do estado (i, x) através de um único movimento válido executado pelo jogador i.

2.5 Encontrando uma estratégia vencedora

Note que esse jogo, tanto na versão original como na versão simplificada, não admite empate. Portanto, um dos jogadores tem uma estratégia vencedora: ou é aquele que começa jogando ou é o segundo a jogar.

Seja $\vec{F}$ o grafo dirigido associado ao vetor de listas `grafo` construido no código acima. Note que, ainda que o grafo representado pelo vetor de listas `grafo_tabuleiro[i]` tenha ciclos (no caso os laços que modelam "passar a vez"), o grafo $\vec{F}$ não tem nenhum circuito, e portanto é um grafo dirigido acíclico em que os nós sorvedouros são precisamente aqueles cuja configuração é terminal.

Dizemos que o vértice $(i, x) \in V(\vec{F})$ possui uma estratégia vencedora se uma das duas condições abaixo estiverem satisfeitas:

- se x é uma configuração terminal onde o jogador i já tenha ganhado e o jogador $1 - i$ não tenha ganhado, ou
- se existe um vértice $(1 - i, y)$ que é vizinho de saída de (i, x) (ou seja, que está na lista `grafo[(i,x)]`) e que não tenha uma estratégia vencedora.

Para descobrir qual dos jogadores possui uma estratégia vencedora, vamos implementar o algoritmo recursivo que corresponde à definição indutiva de estratégia vencedora que foi dada acima.

In [52]:
```
vencedor = {}

# (i, x) not in vencedor.keys() means it was not visited
# vencedor[(i, x)] = 1          means it is being visited
# vencedor[(i, x)] = 2          means it has no winning strategy
# vencedor[(i, x)] = 3          means it has a winning strategy

def ganha(i, x):
    # mark as visited:
    vencedor[(i, x)] = 1
    # no terminal node has a winning strategy:
    if terminal(x):
        vencedor[(i, x)] = 2
        return
```

```
            for j, y in grafo[(i, x)]:
                # Se você implementou o grafo reduzido descrito na seção anterior,
                # então use phi(y) no lugar de y até o final desta função.
                if not (j, y) in vencedor.keys(): ganha(j, y)
                if vencedor[(j, y)] == 2:
                    vencedor[(i, x)] = 3
            if vencedor[(i, x)] == 1:
                vencedor[(i, x)] = 2
```

Decide se existe uma estratégia vencedora para cada possível estado do jogo:

```
In [53]: for i in range(2):
             # Se você implementou o grafo reduzido descrito na seção anterior,
             # então use representantes.keys() no lugar de range(8000) a seguir
             for K in range(8000):
                 if not (i, K) in vencedor.keys():
                     ganha(i, K)
```

Se o jogador 0 começar o jogo, ele tem uma estratégia vencedora?

```
In [72]: vencedor[(0, get_ID(3,3,3))] == 3

Out[72]: True
```

Note que o outro jogador, se ele começar jogando, também tem estratégia vencedora:

```
In [73]: vencedor[(1, get_ID(3,3,3))] == 3

Out[73]: True
```

E que movimento o jogador 0 deve fazer no começo para ganhar não perder o jogo? Bom, a lista dos IDs vizinhos é:

```
In [74]: grafo_tabuleiro[0][get_ID(3,3,3)]

Out[74]: [1267, 1271, 1275, 1343, 1423, 1503, 2863, 4463, 6063]

In [76]: # Quais desses estados vizinhos é bom pro jogador 0?
         for x in grafo_tabuleiro[0][get_ID(3,3,3)]:
             # Aqueles em que o jogador 1 perde!
             if vencedor[(1, x)] == 2:
                 print(x, end=":\n")
                 imprime_config(x)

1271:
..B.P
B...P
B...P

1275:
```

```
...BP
B...P
B...P

1423:
B...P
..B.P
B...P

1503:
B...P
...BP
B...P

4463:
B...P
B...P
..B.P

6063:
B...P
B...P
...BP
```

Portanto qualquer uma das configurações acima é uma boa jogada para o jogador 0. Por outro lado, se o jogador 0 começar indo pra configuração 2863, ele pode perder o jogo (supondo que o jogador 1 jogue perfeitamente depois disso).

```
In [77]: # Portando, seria uma jogada ruim para o jogador 0 fazer uma jogada que
         # levasse o tabuleiro à configuração 2863:
         imprime_config(2863)
```

```
B...P
B...P
.B..P
```

2.6 De que um *bot* precisa para ganhar?

Para que um *bot* seja capaz de jogar esse jogo perfeitamente, ele precisa conhecer a estratégia vencedora para cada estado em que ela existir.

Portanto, para cada estado, ele deve conhecer qual a jogada a ser feita.

Se você implementou o grafo reduzido: basta sabermos essa informação para aquelas configurações que são representantes de grupos; para outra configuração x, podemos ver como jogar a partir de $\Phi(x)$ e depois usar a permutação inversa de π para converter a configuração resposta a uma jogada válida a partir de x.

```
In [85]: import random as rd

         estrategia = [{}, {}]

         for jogador, x in vencedor.keys():
             if terminal(x):
                 continue

             # Parecido com a movimentos_validos_tabuleiro
             v = [col_id[x][i] for i in range(3)]
             jogadas = []
             for i in range(3):
                 u = v.copy()
                 for id in grafo_coluna[jogador][v[i]]:
                     u[i] = id
                     y = get_ID(*u)
                     jogadas.append(y)
             # Verifica se é preciso "passar a vez":
             if len(jogadas) == 0:
                 jogadas.append(x)

             # define estratégia padrão: jogar aleatóriamente
             estrategia[jogador][x] = rd.choice(jogadas)
             for y in jogadas:
                 # mas se uma das jogadas deixa o oponente em posição de derrota...
                 # Se você implementou o grafo reduzido, use phi(y) em vez de y
                 # somente na linha a seguir.
                 if vencedor[(1 - jogador, y)] == 2:
                     # troca a estratégia aleatória por essa jogada
                     estrategia[jogador][x] = y
                     break
```

Repare que um pedaço desse código é bem parecido com aquele que calcula o grafo de movimentos válidos do tabuleiro `grafo_tabuleiro`.

Nota também que, quando não há estratégia vencedora, usamos um movimento válido aleatório.

Para quem implementou `grafo_tabuleiro` usando o grafo reduzido, a principal diferença está no fato de que, quando calculamos a lista de vizinhos de uma configuração, não estamos mais interessados nos representantes dos vizinhos, mas nos prórios vizinhos. Ou seja, `estrategia[jogador][x]` armazena o ID y de uma configuração que é vizinha da configuração x. Isso é importante para podermos descobrir eficientemente qual jogada o *bot* deve fazer em cada configuração. Se tivéssemos armazenado $\Phi(y)$ em vez de y, ainda teríamos o trabalho adicional de descobrir qual coluna deve ser alterada.

```
In [ ]: # Sinal + quer dizer ganhador
        #       - quer dizer perdedor
        print(len(estrategia[0].keys()))
        for i in estrategia[0].keys():
```

```
            if vencedor[(0, i)] == 3:
                print('+', end='')
            else:
                print('-', end='')
            print(i, end=': ')
            if vencedor[(1, estrategia[0][i])] == 3:
                print('+', end='')
            else:
                print('-', end='')
            print(estrategia[0][i])

        # imprime uma lista de jogadas
        # vez do 0 : vez do 1
```

Repare que nem todas as configurações aparecem na tabela. Aquelas que não aparece são as configurações terminais.

```
In [94]: # 7979 não aparece na lista:
         print(terminal(7979))
         imprime_config(7979)
```

```
1
...PB
..P.B
...PB
```

```
In [1]: %%html
        <!-- ESTA CÉLULA SERVE APENAS PARA FORMATAÇÃO DESTE ARQUIVO DE NOTEBOOK -->
        <style>
        .rendered_html h1 {
            background-color: #555;
            color: white;
            padding: .5em;
            // border-bottom: 2px solid #000;
            // padding-bottom: .6em;
            margin-bottom: 1em;
        }

        .rendered_html h1 code {
            color: #EBB7C5;
            background-color: rgba(0,0,0,0);
        }

        .rendered_html h2 {
            border-bottom: 1px solid #333;
            padding-bottom: .6em;
        }
```

```
        .rendered_html h3 {
            color: #034f84;
        }

        .rendered_html code  {
            padding: 2px 4px;
            font-size: 90%;
            color: #c7254e;
            background-color: #f9f2f4;
            border-radius: 4px;
        }

        .rendered_html pre code {
            padding: 0px;
            font-size: 90%;
            color: #c7254e;
            background-color: rgba(0, 0, 0, 0);
        }

        kbd {
            border-radius: 3px;
            padding: 2px, 3px;
        }

        body {
            counter-reset: h1counter excounter;
        }
        h1:before {
            content: counter(h1counter) ".\0000a0\0000a0";
            counter-increment: h1counter;
        }
        span.exec:before {
            content: counter(excounter);
            counter-increment: excounter;
        }


        </style>
```

```
<IPython.core.display.HTML object>
```