Universidade Federal do Rio Grande do Sul **Inteligência Artificial**
Instituto de Informática **Prof. Joel Luís Carbonera**
Departamento de Informática Aplicada

Trabalho 4
Busca com Adversário

Instruções preliminares

Este trabalho envolve a implementação de algoritmos de busca com adversário em dois jogos. Um kit com os arquivos necessários para executar o trabalho está disponível no moodle (arquivo `kit_games.zip`).

As implementações devem ser feitas em Python 3. Assuma que os códigos serão executados em uma máquina com interpretador Python 3.12, com Miniconda, Pip, numba, numpy, pandas). Caso precise de instalar bibliotecas adicionais, descreva-as no seu `Readme.md`

1. Os Jogos

Você implementará algoritmos para dois jogos: jogo da velha invertido (tic-tac-toe misere) e Othello (também conhecido como Reversi). Cada jogo é implementado em dois módulos (`board.py` e `gamestate.py`) em um pacote no kit. Jogo da velha invertido está no pacote `tttm` (abreviatura para tic-tac-toe misere) e Othello está no pacote `othello`.

O `gamestate.py` contém uma classe `GameState`, cujos objetos armazenam o tabuleiro no atributo `board` e o jogador a fazer a jogada no atributo `player` (um caractere, `B` para as pretas ou `W` para as brancas. Isso vale para ambos os jogos (jogo da velha invertido também é jogado com 'pretas' ou 'brancas' ao invés dos clássicos 'X' e 'O'). As funções de gamestate.py permitem consultar a lista de ações válidas, obter a lista de sucessores de um estado, consultar se o estado é terminal e quem venceu o jogo.

O `board.py` contém a classe `Board`, cujos objetos contém a representação interna do jogo. Ela somente será útil para calcular avaliações heurísticas na qual você precisará examinar o estado internamente.

1.1. Jogo da velha invertido

O tamanho do tabuleiro e as jogadas válidas são da mesma forma que na versão original, porém, no invertido você **PERDE** ao alinhar 3 peças em um grid 3x3. Este é um jogo pequeno: o fator de ramificação começa em 9 e diminui em 1 a cada turno. A profundidade máxima da árvore do jogo é 9.

A seguir temos uma descrição mais detalhada do `tttm/board.py`. O tabuleiro é representado como uma matriz de caracteres (ou lista de strings ;). `W` representa uma peça branca (white), `B` uma peça preta (black) e `.` (ponto) representa um espaço livre. No exemplo a seguir, temos a representação de um estado de vitória das pretas (brancas alinharam 3 peças na diagonal).

```
[
“..W”,
“BWB”,
“WB.”
]
```

Em nosso sistema de coordenadas, eixo x cresce da esquerda para a direita e o eixo y cresce de cima para baixo. O exemplo a seguir mostra o sistema de coordenadas para aquele mesmo estado acima.

```
  012 -> eixo x
0 ..W
1 BWB
2 WB.
|
v
eixo y
```

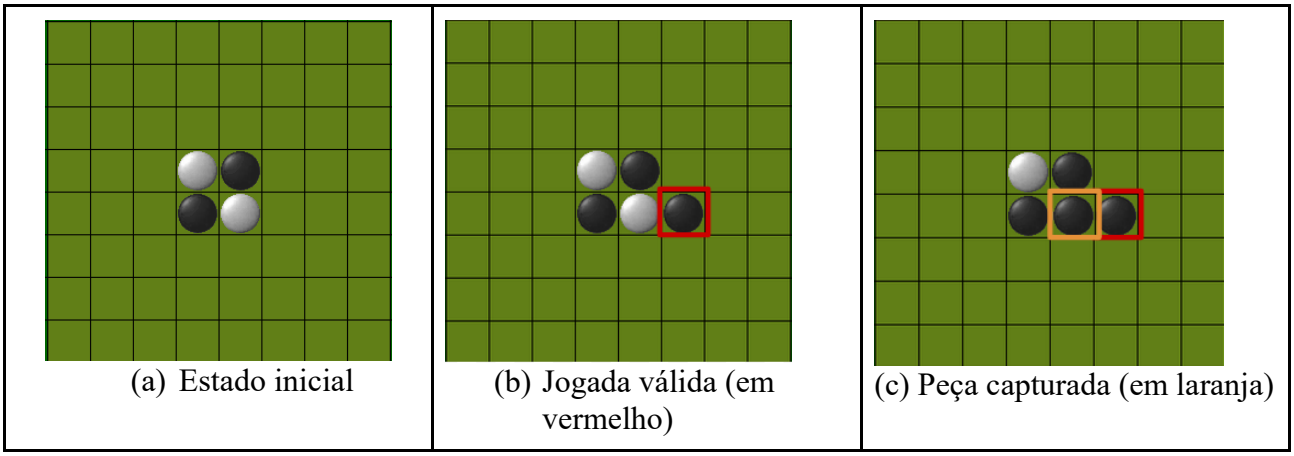
Uma jogada é dada por uma tupla com as coordenadas x, y (coluna, linha) da peça a ser colocada, de acordo com o sistema de coordenadas acima.

Para praticar o jogo e o sistema de coordenadas, use o servidor do kit com o comando a seguir, que iniciará uma partida em que ambas as cores são controladas via terminal:

```
python server.py tttm advsearch/humanplayer/agent.py
advsearch/humanplayer/agent.py
```

1.2. Othello

O tabuleiro é um grid de 8x8 posições cujas células centrais começam preenchidas por duas peças brancas e pretas, conforme a Figura 1 (a).



(a) Estado inicial

(b) Jogada válida (em vermelho)

(c) Peça capturada (em laranja)

Figura 1 - Jogo de Othello

As pretas começam jogando e, em cada turno, um jogador deve colocar a próxima peça somente em

posições onde uma peça adversária seja capturada. Uma ou mais peças do adversário são capturadas quando existe uma linha reta – horizontal, vertical ou diagonal – entre a peça colocada pelo jogador e uma das suas outras peças. Todas as peças capturadas mudam de cor e o jogo continua alternadamente. Caso um jogador não possua uma jogada válida, ele passa a vez. O jogo termina quando não houver jogadas válidas para ambos os jogadores ou quando não restarem peças de um jogador. O vencedor é aquele com o maior número de peças no tabuleiro.

O link http://en.wikipedia.org/wiki/Reversi possui mais detalhes sobre regras, história, etc.
Neste site você consegue jogar reversi contra um algoritmo de IA.

A seguir temos uma descrição mais detalhada do `othello/board.py`. Em um objeto da classe `Board`, o atributo `tiles` contém a representação do tabuleiro como uma matriz de caracteres (ou lista de strings ;). `W` representa uma peça branca (white), `B` uma peça preta (black) e `.` (ponto) representa um espaço livre. No exemplo a seguir, temos a representação do estado inicial da Figura 1 (a).

```
[
“........”,
“........”,
“........”,
“........”,
“...WB...”,
“...BW...”,
“........”,
“........”,
“........”,
]
```

O eixo x cresce da esquerda para a direita e o eixo y cresce de cima para baixo. O exemplo a seguir mostra o sistema de coordenadas para o estado inicial.

```
  01234567 -> eixo x
0 ........
1 ........
2 ........
3 ...WB...
4 ...BW...
5 ........
6 ........
7 ........
|
v
eixo y
```

Uma jogada é dada por uma tupla com as coordenadas x, y (coluna, linha) da peça a ser colocada, de acordo com o sistema de coordenadas acima.

Para praticar o jogo e o sistema de coordenadas, use o servidor do kit com o comando (iniciará uma partida em que ambas as cores são controladas via terminal):

```
python server.py othello advsearch/humanplayer/agent.py
advsearch/humanplayer/agent.py
```

2. O algoritmo

Você deverá implementar o minimax com poda alfa-beta. Opcionalmente, você pode implementar também o MCTS como extra. Os algoritmos devem ser implementados para ambos os jogos. Todas as implementações consistem em preencher as funções especificadas em determinados arquivos do kit.

2.1. Minimax com poda alfa-beta

Você deve implementar a poda alfa-beta na função `minimax_move` do arquivo `minimax.py`. A função recebe o estado do jogo, a profundidade máxima (-1 pra profundidade ilimitada) e uma função de avaliação (que você implementará em um momento posterior. Ela deve executar a poda alfa-beta até a profundidade máxima e chamar a função de avaliação quando encontrar um estado terminal ou nos estados na profundidade máxima.

Sua implementação da poda alfa-beta deve ser independente do jogo (ambos os jogos usam a mesma interface para consultar as jogadas e gerar os sucessores). A única parte que depende do jogo específico é a função de avaliação de estados, a qual é implementada em outro arquivo, conforme a seguir. Todas as funções de avaliação recebem o estado a ser avaliado e o jogador cujo ponto de vista deve ser considerado. Esse jogador deve ser o mesmo do estado raiz da árvore de busca, já que a decisão será feita para ele.

2.1.1. Jogo da velha invertido

Para o jogo da velha invertido, preencha a função `utility` do arquivo `tttm_minimax.py` com a função de avaliação. Assuma que ela avaliará estados terminais do jogo. Por fim, a função que retornará uma jogada para o servidor de jogos é a `make_move`. Ela receberá um estado e executará a poda alfa-beta sem limite de profundidade, passando sua função `utility` para avaliação dos estados. O jogo é pequeno e é esperado que seja possível atingir a profundidade máxima. Quando testada, sua função deve retornar uma jogada com o tempo limite de 1 minuto.

2.1.2. Othello

Othello é um jogo mais complexo, com fator de ramificação variável, mas com profundidade máxima de 64, o que torna inviável resolvê-lo completamente. Você implementará 3 heurísticas em arquivos diferentes.

(obrigatório) Em cada arquivo, você deverá preencher o `make_move` para executar sua poda alfa-beta com profundidade limitada e a respectiva heurística de avaliação implementada naquele mesmo arquivo. Você deve definir a profundidade máxima para retornar jogadas no tempo limite de 5 segundos. As 3 heurísticas são descritas a seguir.

(opcional) Você pode adicionar melhorias para a poda alfa-beta: por exemplo, busca com aprofundamento iterativo, etc. É possível pesquisar e implementar técnicas não vistas em aula. Neste caso, a técnica deve ser descrita no relatório entregue.

2.1.2.1. Heurística da Contagem de peças

Essa heurística simplesmente retorna a diferença entre a quantidade de peças do jogador e do seu oponente. A poda alfa beta com essa heurística deve ser implementada na função `evaluate_count` do arquivo `othello_minimax_count.py`. O `make_move` desse arquivo deve chamar sua poda alfa-beta usando `evaluate_count` como função de avaliação.

2.1.2.2. **Heurística do Valor posicional**

Essa heurística atribui valores para posições do tabuleiro (e.g. quinas valem muito, pré-quinas valem pouco) e retorna a diferença de valores das posições ocupadas pelo jogador e o oponente. A poda alfa beta com essa heurística deve ser implementada na função `evaluate_mask` do arquivo `othello_minimax_mask.py`. O valor das posições é uma matriz pré definida que serve como máscara para o tabuleiro. Não mude esses valores. O `make_move` desse arquivo deve chamar sua poda alfa-beta usando `evaluate_mask` como função de avaliação.

2.1.2.3. **Heurística customizada**

Elabore uma heurística com desempenho **melhor** do que as anteriores. Não vale qualquer heurística, ela deve ser melhor. A poda alfa beta com essa heurística deve ser implementada na função `evaluate_custom` do arquivo `othello_minimax_custom.py`. O `make_move` desse arquivo deve chamar sua poda alfa-beta usando `evaluate_custom` como função de avaliação.

2.2. Avaliação

a) Para o Tic-Tac-Toe misere, relate se o desempenho da sua implementação do minimax com poda alfa-beta. Embora não seja simples provar que o jogo está sendo jogado com perfeição, avalie as evidências:

(i) O minimax sempre ganha ou empata jogando contra o randomplayer?
(ii) O minimax sempre empata consigo mesmo?
(iii) O minimax não perde para você quando você usa a sua melhor estratégia?

Relate a resposta para cada um dos itens no relatório.

b) Para o Othello, faça um mini-torneio entre os algoritmos (minimax com as três heurísticas), considerando as partidas abaixo. Para cada partida, relate quem venceu e o número de peças final de cada agente. Note que os mesmos oponentes se enfrentam duas vezes, em cada uma delas, cada oponente começa jogando uma vez.

Partidas:
Contagem de peças X Valor posicional:
Valor posicional X Contagem de peças:
Contagem de peças X Heurística customizada:
Heurística customizada X Contagem de peças:
Valor posicional X Heurística customizada:
Heurística customizada X Valor posicional:

Também observe e relate qual implementação foi a mais bem-sucedida de todas (a que mais teve vitórias e, caso tenha empate nesse critério, a que mais capturou peças).

(opcional) Você pode implementar o MCTS e incluir ele no torneio, implementando técnicas não vistas em aula, se quiser.

IMPORTANTE: cuidado com o sistema de coordenadas vs a indexação de matrizes. Sua função make_move deve retornar as coordenadas `x, y` (coluna, linha) enquanto a representação de matriz endereça primeiramente a linha e depois a coluna.

3. Torneio de Othello

Haverá um torneio de Othello entre os programas dos estudantes, em formato a ser definido. Dadas as suas observações, preencha o make_move do arquivo tournament_agent.py para retornar a jogada. A estratégia pode usar a sua melhor implementação no exercício anterior, ou ter ainda outras melhorias, ou usar MCTS. Necessariamente, se sua melhor implementação for baseada em minimax, ela deve usar a heurística costumizada (ou uma melhor), e pode incluir melhorias em relação ao minimax clássico com poda alfa beta. Note que ela não deve utilizar puramente nenhuma das heurísticas básicas do enunciado, já que é esperado que as heurísticas dos oponentes no torneio sempre serão melhores. O player do torneio deve ser baseado em minimax ou MCTS, mas pode utilizar outras técnicas complementares sofisticadas. Durante cada partida do torneio, sua função `make_move` será chamada diversas vezes, uma para cada jogada a ser feita pelo seu jogador. Por isso, a função não deve gerar novos processos ou threads que rodem em background após seu término, sob pena de desclassificação no torneio. **A função terá tempo limite de 5 segundos para executar e não pode consumir mais que 4 Gb de RAM**. Se utilizar mais tempo ou mais memória, o agente é desclassificado do torneio. Note que a implementação também não pode armazenar informações em arquivos externos ou outras informações pré-calculadas no código (por exemplo, pré-calcular algumas jogadas com algum algoritmo externo e incluir essas informações no código). Isso é considerado doping e gera desclassificação. Antes do torneio, será realizada uma pré-seleção para verificar quais agentes estão aptos para o torneio. Se o agente for desclassificado por algum motivo (não segue especificações, gera muitas violações de tempo, consome muita memória, utiliza artifícios proibidos,...), o grupo perde 10% da nota do trabalho. Ou seja, a ideia é que todos os grupos participem.

4. Entrega

O kit do trabalho contém um diretorio "`your_agent`", no qual você deve fazer sua implementação. Renomeie `your_agent` com o nome do seu agente. Use as convenções e nomes de pacotes de python, já que seu agente é visto pelo servidor de partidas como um pacote.

Você pode criar arquivos e pacotes auxiliares dentro do seu diretório, mas você deve preencher as funções dos arquivos especificados, já que elas serão testadas. Não mude a assinatura (parâmetros e retorno) das funções. Isso poderá inviabilizar a execução e/ou a correção das mesmas.

Ao final, gere um arquivo `.zip` contendo sua implementação e um arquivo `Readme.md` com um relatório da sua implementação. O relatório deve conter:

- Nomes, cartões de matrícula e turma dos integrantes do grupo;
- Bibliotecas que precisem ser instaladas para executar sua implementação;
- Resultado da sua avaliação da poda alfa-beta no tic-tac-toe misere (ver item "a" seção 2.3);
- Para o Othello:
  - Explique a heurística customizada e, caso tenha sido utilizada alguma fonte (como artigo ou site), indique a fonte também, explicando como as fontes foram utilizadas (a heurística foi utilizada conforme apresentada na fonte, foi uma combinação de ideias de fontes diferentes, foi totalmente projetada pelo grupo, sem utilização de fontes,...);
  - descrição do critério de parada do agente (profundidade máxima fixa? aprofundamento iterativo parado por tempo?etc);
  - Resultado da avaliação (ver item "b" da seção 2.3);
  - Explique a implementação escolhida para o torneio.
  - Extras: Relate qualquer item opcional (como implementação do MCTS) ou melhoria não mencionada (técnicas adicionais para melhorar o minimax não vistas em aula) que

você tenha realizado e, caso tenha utilizado fontes extras para auxiliar, mencione as fontes e como foram utilizadas.

Observação: É encorajado que os grupos consultem outras fontes, se quiserem e não há nenhum problema caso técnicas eu heurísticas sejam utilizadas tais como estão nas fontes. ChatGPT também vale 😊

Supondo que o grupo renomeou o diretorio `your_agent` do kit como `alan_turing`, a estrutura do arquivo `.zip` a ser entregue deve ser a seguinte:

```
alan_turing <-- diretorio na raiz do .zip
|-- __init__.py
|-- mcts.py              <-- implementação do MCTS (opcional)
|-- minimax.py           <-- implementação da poda alfa-beta
|-- othello_minimax_count.py   <-- heuristica de contagem
|-- othello_minimax_custom.py  <-- heuristica customizada
|-- othello_minimax_mask.py    <-- heuristica posicional
|-- tournament_agent.py  <-- melhor agente pro torneio de Othello
|-- tttm_minimax.py    <-- minimax que joga o tic-tac-toe misere
|-- Readme.md <-- com seu relatorio
\-- [outros arquivos e subdiretorios de sua implementacao]
```

Sua implementação será executada na mesma estrutura de diretórios do kit do trabalho.

5. Critérios de correção

Item	**Percentual da nota**
Aderência à especificação (e.g. estrutura de diretorios, protocolo de jogadas)	10
Implementação do minimax com poda alfa-beta	40
Relatório	
Item a da seção 2.2	15
Item b da seção 2.2	35
Extras	10
Total	**110**

Importante:

- O trabalho deve ser feito em grupos.
- O tempo de **5 segundos** é estipulado tendo como referência uma máquina Windows com a seguinte configuração: **processador Xeon E5-2650 Sandy Bridge (Q1'12), 2,0 GHz e 4Gb de memória RAM** (possivelmente essa será a configuração do computador do torneio).

- A nota depende do correto funcionamento da implementação e de um bom relatório. Isto é, um mau desempenho no torneio não resultará em penalidade na nota (desde que seu agente respeite o protocolo e não seja desclassificado).

Dicas

- Em algumas ocasiões, o mesmo jogador pode jogar duas (ou mais) vezes seguidas, pois o oponente fica sem jogadas. Ajuste sua implementação da poda alfa-beta (e MCTS, se implementar) para verificar e tratar isso apropriadamente.
- Leia o `README.md` do `kit_games.zip`, ele contém instruções para a execução do servidor e do jogador 'random'.
- Você pode usar as funções do `gamestate.py` para gerar a lista de jogadas válidas e os estados resultantes das mesmas.
- Você pode aproveitar o servidor de partidas para iniciar o seu agente a partir de um estado que esteja causando erros (você escreve a representação com o estado problemático e executa seu agente).