TP1: Othello
Deep Diver

André Harder (2011048910) *andreharder@dcc.ufmg.br*
Marzo Torres Junior (2012422670) *marzojr@dcc.ufmg.br*

1 Introdução

Othello é um jogo que foi criado no Japão na década de 1970, tendo sido baseado em um jogo chamado *Reversi* que foi criado em 1883 na Inglaterra. O *Othello* é um jogo de tabuleiro, sendo que a versão mais usada usa um tabuleiro de 8x8 casas e um conjunto de discos pretos e brancos. As regras são simples:

- O *Othello* começa com as quatro casas centrais preenchidas alternadamente por discos pretos e brancos;
- O jogador com os discos pretos tem o primeiro lance; os jogadores alternam a cada lance;
- Um lance válido corresponde a escolher uma casa vazia tal que:
  - deve existir pelo menos uma sequência contígua de discos em linha reta (horizontalmente, verticalmente, ou em diagonal) que termina adjacente a esta casa vazia;
  - esta sequência deve ter pelo menos um disco do jogador corrente, e pelo menos um disco do outro jogador, sendo que pelo menos um destes últimos deve ser ajacente à casa vazia selecionada.

  Ao ser realizado, o jogador coloca um disco seu na casa selecionada e troca todos discos do oponente que obedecem as condições acima por discos seus;
- Na ausência de lances válidos de um jogador, o outro tem a vez;
- O jogo termina quando nenhum jogador tem lances válidos, com o ganhador sendo o que tiver mais peças ao final.

O interesse do *Othello* para inteligência artificial vem do fato que ele ainda não é um jogo resolvido: as versões com tabuleiros de 4x4 e 6x6 casas já foram resolvidos, mas o tabuleiro 8x8 ainda apresenta desafios computacionais para resolver. Assim, há um grande interesse no desenvolvimento de heurísticas para avaliar posições e levar a jogadores melhores.

O resultado do nosso trabalho o *"Deep Diver"*, um jogador de *Othello* baseado no algoritmo *MiniMax* com poda $\alpha - \beta$, assim chamado porque consegue fazer buscas em profundidade 8 no tempo requisito de 5 segundos.

2 Função de avaliação

Foram empregadas 5 formas de avaliar um estado, as quais são unidas em uma única métrica por via de uma combinação linear. Descrevemos aqui cada uma das mesmas, tal como a metodologia utilizada para parametrizar os escalares aplicados a cada um deles.

2.1 Fim de jogo

Verifica se nenhum dos jogadores possui jogadas válidas, designando o fim do jogo. Sendo este o caso, a heurística retorna um valor grande (ordens de magnitude maior que os valores das outras heurísticas) e proporcional ao número de peças que o jogador avaliado possui em jogo (desta forma, prefirindo vitórias com mais peças à vitórias com poucas peças).

É a avaliação mais direta utilizada, que visa evitar que, no fim do jogo, a IA opte por, por exemplo, pegar um canto (que provê um valor heurístico alto) ao invés de ganhar o jogo.

2.2 Diferença no número de peças

Computa-se o número de peças de cada jogador, e retorna-se a diferença. Como esta métrica não é normalizada, seu impacto é inicialmente pequeno, porém cresce linearmente com o decorrer do jogo, incentivando a ocupação de posições mais estratégicas no inicio (tal como a evaporação [sessão 2.4]).

2.3 Peso posicional

Definimos pesos para cada posição no tabuleiro, e para cada uma das mesmas, se o jogador que deseja-se otimizar possui uma peça ali, soma-se o peso associado. A métrica é então normalizada, subtraindo do valor da soma o valor da soma do oponente, e dividindo pela soma de ambos (ou, caso esta soma seja 0, usa-se 0).

A tabela empregada é a seguinte (originalmente de [2]):

99	-8	8	6	6	8	-8	99
-8	-24	-4	-3	-3	-4	-24	-8
8	-4	7	4	4	7	-4	8
6	-3	4	0	0	4	-3	6
6	-3	4	0	0	4	-3	6
8	-4	7	4	4	7	-4	8
-8	-24	-4	-3	-3	-4	-24	-8
99	-8	8	6	6	8	-8	99

Sendo a função de avaliação e a normalização descritas por:

$$
\begin{aligned}
b_{ij}^{(K)} &= \begin{cases} 1 & \text{se o jogador } K \text{ tiver uma peça em } (i,j) \\ 0 & \text{caso contrário} \end{cases} \\
w &: \text{matriz de pesos} \\
\text{Heurística} &= \frac{B^{(J)} \cdot W - B^{(\neg J)} \cdot W}{B^{(J)} \cdot W + B^{(\neg J)} \cdot W} \\
J &: \text{Jogador que está sendo otimizado}
\end{aligned}
\tag{1}
$$

2.4 Fronteira relativa

A heurística aqui considera o número relativo de jogadas que o jogador a ser otimizado tem com relação ao outro. Deseja-se aumentar este valor pois espera-se que (1) tendo mais lugares para se colocar peças é mais provável haver uma estratégia boa, e (2) reduzindo o número de jogadas possíveis do oponente, este será obrigado a tomar ações indesejáveis.

A função de avaliação é descrita por:

$$
\begin{aligned}
\text{Heurística} &= \frac{\text{Jogadas possíveis de } J - \text{Jogadas possíveis de } \neg J}{\text{Jogadas possíveis de } J + \text{Jogadas possíveis de } \neg J} \\
J &: \text{Jogador que está sendo otimizado}
\end{aligned}
\tag{2}
$$

2.5 Proximidade de Canto (*Corner Closeness*)

Esta heurística tem o propósito de desencorajar o Deep Diver de escolher movimentos que levem ele a ter discos nas casas adjacentes ao canto sendo que o canto em si está livre. Estas configurações tendem a ser ruins porque elas tem alta chance de ceder o canto para o oponente.

2.6 Parametrização

Utilizou-se uma estratégia de combinação linear para determinar a influência de cada métrica. Para fazer a parametrização, utilizou-se uma técnica de subida de gradiente (*hill climbing*) com múltiplos inícios aleatórios (em particular, 16 seeds, que é o número de processadores no *cluster* usado para os testes).

Nesta técnica, afixamos o peso da primeira heurística (diferença no número de peças – o condicional de fim de jogo manteve peso constante) em 0.05, e usamos valores aleatóriamente gerados entre 0 e 1 para as demais (0.05 é um valor arbitrário – afixamos ele por se tratar de uma combinação linear, onde pode-se afixar um valor sem impactos no resultado).

A variação foi feita um parâmetro por vez, em incrementos de 5%, usando o seguinte procedimento:

1. Crie dois candidatos para o novo valor, um 5% acima e o outro 5% abaixo do valor corrente.

2. Crie duas novas heurísticas cada qual usando um dos candidatos (e os outros parâmetros, que se mantém inalterados), e compute duas partidas entre elas, uma em que o primeiro começa, e outra em que o segundo começa.

3. Se houve empate, torne a aumentar/reduzir cada candidato em mais 5%, e retorne a etapa 2). Caso contrário, prossiga:

4. Usando a heurística vitoriosa do passo anterior e a heurística original, jogue mais dois jogos (trocando o jogador inicial em cada caso), e se houver empate, use o novo valor computado, caso contrário, mantenha o vencedor.

Note que o uso do novo valor, no caso do empate, foi adotado para aumentar a exploração do espaço de busca.

2.7 Estratégia de parada

O Deep Diver tem três estratégias para parar a busca:

1. busca limitada a profundidade 8;

2. Se a busca até o momento gastou mais do que 4 segundos, a profundidade passa a ser limitada a 6 níveis;

3. uma folha (posição na qual nenhum jogador tem movimentos) foi encontrada.

Quando os casos 1 e 2 ocorrem, o *MiniMax* retorna o valor estimado do tabuleiro corrente segundo a função de avaliação. No caso 3, o valor retornado é o valor final do tabuleiro, multiplicado por um fator grande.

O caso 2 foi adicionado, em grande parte, para garantir que o Deep Diver não vai exceder o limite de tempo especificado; na prática, ele quase nunca demora mais do que 5 segundos para pesquisar a profundidade 8.

3 Melhorias

Algumas possíveis melhorias que foram consideradas, mas acabaram por não ser implementadas:

3.1 Tabelas de Abertura e Fim de Jogo

O *Deep Diver* utiliza 33 jogadas pré-definidas seguindo as 77 aberturas mostradas em [2][1], com algumas pequenas adaptações para tornar-las utilizáveis na implementação. Estas jogadas são aberturas, adentrando até no máximo a 22a jogada.

[1]Muitas das sequencias são subsequencias de outras, logo há mais aberturas do que pares tabuleiro/jogadas

Um conjunto mais extenso de tabelas pode melhorar ainda mais o desempenho do *Deep Diver*.

3.2 Tabela de Lookup

Após a execução de um *profiler* no código, constatou-se que a função de expansão de nós (que computa as jogadas possíveis a partir de uma determinada configuração do tabuleiro) gastava pouco menos de 80% do tempo de execução do código.

Visando mitigar este custo, optou-se pelo uso de uma lookup table (parcial) para computar as posições possíveis onde poderia-se inserir uma nova peça.

Sem entrar em pormenores detalhes, a estratégia adotada foi de dividir o tabuleiro em 28 segmentos (8 + 8 horizontais/verticais e 11 + 11 diagonais de tamanho maior que 2), e então para cada tamanho de segmento (tamanhos de 3 a 8) mapear todas as combinações de peças possíveis para as posições vazias nelas que poderiam comportar uma nova peça (para cada uma das duas cores). Computamos então este conjunto para cada segmento, deslocando as posições ocupáveis retornadas pela *lookup table* para suas posições apropriadas, e então tomamos a união, finalmente obtendo com isso todas as posições onde pode-se inserir uma peça branca ou preta.

O resultado conjunto desta otimização e do abandono do *std::vector* em prol de *pointer-arrays* resultou em um speedup de $\sim 50X$, reduzindo o tempo gasto para ir a uma profundidade 10 no *Minimax* de 82 para 1.5 segundos (este tempo foi tirado para uma instância na maquina de de teste; Considerando outros casos com um branching factor mais alto, a lerdeza das maquinas de referência e a margem de segurança, optou-se por usar uma profundidade 8).

3.3 Busca mais profunda em fim-de-jogo

O Deep-diver realiza buscas mais aprofundadas no fim-de-jogo; especificamente, ele sempre procura até o fim da árvore de busca sempre que existam pelo menos 52 peças no tabuleiro. Isto permite um fim-de-jogo mais preciso. Nos testes realizados nos laboratórios, isco acaba nunca demorando mais do que 2 segundos por lance nestes casos, e permite avaliar o resultado da busca de forma mais eficiente.

3.4 Melhorias consideradas, porém não implementadas

3.4.1 Ordenamento de lances

Uma outra possibilidade de melhoria seria ordenar os lances segundo alguma heurística (possivelmente as mesmas heurísticas usadas na função de avaliação) para tentar cair em casos melhores para a poda $\alpha - \beta$. Uma tentativa que foi realizada foi ordenar os movimentos pelo valor da heurística em uma busca rasa (2 níveis de profundidade); esta tentativa não resultou em melhoras (pelo contrário, tornou a pesquisa mais lenta), e acabou sendo descartada.

3.4.2 Quiescent search, singular extensions

Estas estratégias foram consideradas para mitigar o efeito horizonte. No final, o Deep Diver acaba quase sempre terminando a busca em um lance do jogador min, de modo que se o último lance do jogador max pode ser eficientemente contra-atacado, a busca vai ver o contra-ataque. Por fim, optamos tentar otimizar as heurísticas na função de avaliação.

4 Decisões do projeto e Dificuldades

Desde o princípio o projeto foi concebido para ser em *C++* para ter flexibilidade e abstração sem perda de desempenho, restrito ao subconjunto comum com C quando necessário para o melhor desempenho possível.

Boa parte da discussão inicial foi em torno da representação interna de uma configuração do tabuleiro; diversas alternativas foram propostas e examinadas, até que foi decidida que a representação mais simples (um byte para cada casa do tabuleiro) provavelmente exigiria menos esforço computacional para usar, levando a um melhor desempenho.

O Deep Diver é altamente otimizado. Algumas das estratégias usadas para melhorar o desempenho são:

- O uso de *templates* na função *MiniMax*: isto permite ter duas versões distintas, uma que maximiza e outra que minimiza, sem necessitar duplicação de código.

- O uso de *lookup tables* geradas por computador na função expande: após a versão inicial ter sido escrita, foi constatado usando um profiler que a maior parte do tempo era gasto nesta função; por conseguinte, ela foi otimizada usando tabelas para determinar se movimentos são possíveis ou não.

- O uso de tabelas Hash na tabela de aberturas: embora o tamanho das tabelas acabou sendo pequeno, mesmo uma busca com $O(log(N))$ acabaria envolvendo diversas comparações de tabuleiros, o que reduziria o tempo disponível para busca no *MiniMax*.

Como foi usado *C++*, foi possível realizar estas otimizações sem perda significativa da capacidade de abstração, manutenibilidade ou legibilidade do código. No caso, foram usadas diversas capabilidades do padrão novo do *C++*; isto limita a seleção de máquinas a aquelas com *GCC* versão 4.6 ou superior. Isto corresponde a todas máquinas *GNU/Linux* nos laboratórios 2020 e 2021, e todas máquinas *GNU/Linux* do laboratório 2019 exceto pelas máquinas cipo.grad, guaxupe.grad e ganges.grad.

4.1 Testes

A implementação foi primeiro testada durante a sua implementação por via de um conjunto de tabuleiros de teste, que retratam casos extremos onde o fun-

cionamento poderia ter problemas.

Após estarmos satisfeitos quanto ao funcionamento correto do trabalho, testamos ele (1) contra si mesmo, usando uma implementação interna (a qual foi usada para fazer a parametrização descrita na sessão 2.6), (2) contra a IA aleatória, usando o servidor providenciado, e (3) contra o campeão do torneio de reversi do semestre anterior (pelo qual agradecemos *Túlio Loures* e *Yuri Pessoa* por ter disponibilizado o código).

Os resultados inicias desta execução mostraram o *deep diver* definitivamente melhor que o aleatório, e tão bom quanto o *othulio* (o campeão anterior). Depois de uma pequena reparametrização (usando um segundo ótimo encontrado pelo *hill climbing*), o *deep diver* consegue sempre ganhar do *othulio* se ele começa, e empatar caso contrário.

5 Conclusão

Neste trabalho, objetivou-se construir um agente para jogar *Othello* usando o *MiniMax* com poda $\alpha - \beta$ que, dado as restrições de tempo, tivesse uma performance boa em competição com outros agentes.

Para tal, realizamos pesquisas sobre heurísticas a serem usadas, e aplicamos meta-heurísticas para otimizar os pesos relativos das heurísticas na função de avaliação.

Visando melhorar a performance, fizemos uma serie de otimizações guiadas por *profiling* do código que permitiu identificar gargalos e no código e otimiza-los, de modo a tornar o *Deep Diver* mais rápido e capaz de pesquisar mais a fundo do que (espera-se) os potenciais concorrentes conseguirão no mesmo intervalo de tempo.

Acreditamos que o trabalho tenha tido sucesso, porém somente teremos resposta real a isto após a competição.

Referencias

[1] SJ Russell, P. Norvig; Artificial Intelligence: A Modern Approach. (3rd Edition) Prentice Hall (2003)

[2] Estratégias e aberturas para o Reversi
`http://www.samsoft.org.uk/reversi/strategy.htm`
`http://www.samsoft.org.uk/reversi/openings.htm`

[3] Mais estratégias Para o Reversi
`http://home.datacomm.ch/t_wolf/tw/misc/reversi/html/index.html`

[4] Heurísticas para o Reversi
`http://kartikkukreja.wordpress.com/2013/03/30/heuristic-function-for-reversiothello/`

https://github.com/kartikkukreja/blog-codes/blob/master/src/HeuristicFunctionforReversi(Othello).cpp

[5] Algorítmo *Minimax Alpha-Beta*
http://en.wikipedia.org/wiki/Minimax
http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning