Relatório do Agente jogador de Othello

Guilherme Freire, Mateus Ildefonso, Matheus Andrade

Agente 1

Para o primeiro agente, pensamos em usar o minimax padrão. A implementação do algoritmo principal não tem muito mistério, fizemos uma busca em profundidade limitada. A partir do último nível, determinamos os valores dos níveis superiores, sempre alternando entre a escolha do mínimo e do máximo. Até que chegamos à raiz, que escolherá a melhor jogada baseado no que chegou até ela.

Resta então formular uma função de avaliação do tabuleiro. Para isso, usando conhecimento sobre o jogo, não escolhemos "o maior número de peças da minha cor" como heurística. Essa estrategia é notoriamente ruim e só daria certo se a busca da árvore fosse completa. O Othello é um jogo de muita instabilidade nas peças, a cada movimento uma quantidade enorme de peças pode mudar de cor. Por isso devemos usar uma estrategia mais inteligente.

Pesquisando um pouco na internet, descobrimos um tabuleiro no qual cada casa tem um valor numérico. A função heurística é dada pela soma dos valores das casas que cada jogador tem. Esses valores nas casas do tabuleiro funcionam como pesos, e foram calculados considerando o quão bom é ter certa casa ao longo do jogo.

Usando essa estratégia já obtivemos um resultado bem melhor, e fomos capaz de derrotar o "corner_player" e "random_player" com facilidade. Entretanto a busca ainda era raza demais para ser realmente efetiva contra outros bots jogadores de Othello encontrados na internet. Com isso em mente, partimos para o agente 2.

Agente 2

O agente dois é um pouco mais robusto do que o primeiro. Para começar, ele utiliza o corte Alpha-Beta para diminuir o espaço de busca e por tanto poder explorar níveis mais profundos da árvore.

Explorando mais níveis, garantimos que vamos ter um resultado mais exato do valor real de um dado tabuleiro, portanto, a escolha feita pelo algoritmo será mais adequada.

Além do corte Alpha-Beta, implementamos também a análise de quietude de uma dada configuração. Como visto, em sala, o valor de um tabuleiro pode variar muito dependendo de quantos níveis analisamos da árvore. Para evitar isso, verificamos o quanto o valor de um tabuleiro muda, de acordo com a exploração do próximo nível da árvore. Se essa mudança passar de um *threshold*, significa que essa configuração tem uma avaliação instável. Tendo uma avaliação instável, vale a pena explorarmos mais um nível para garantirmos uma avaliação mais precisa desse estado.

Além disso, o Agente2 tira proveito do fato de a árvore de busca ficar cada vez menor conforme o jogo vai terminando e modifica a sua profundidade. Com essa profundidade adaptável, o Agente2 consegue jogar rapidamente e precisamente. Quando o jogo chega próximo do fim, o agente aumenta o nível da sua profundidade de busca, para garantir melhores resultados.

Outras Ideias

Tivemos muitas ideias ao longo do trabalho, entretanto devido a uma questão de tempo (e disponibilidade para trabalhar nesse projeto, prejudicada por trabalhos de outras matérias) não conseguimos desenvolvê-las em código.

A primeira ideia promissora foi usar uma rede neural para jogar Othello. A rede receberia o tabuleiro como entrada, ou seja, um vetor de 64 posições com possíveis valores 1 = jogador branco, 0 = casa vazia e -1 = jogador preto.

Para treinar a rede usaríamos uma abordagem genética notória por bom desempenho em diversas áreas chamada NEAT. NEAT significa Neural Evolution of Augmenting Topologies e é uma forma de codificação de redes neurais para algoritmos genéticos. Ela se destaca principalemente por conseguir evoluir tanto a topologia da rede quanto os pesos de cada aresta.

Sua codificação pode ser melhor explorada no artigo original. Mas resumidamente, o DNA da rede é separado em dois grupos: nós e arestas. O grupo dos nós apenas guarda o índice do nó e um valor chamado de "inovação" - utilizado para o crossover - para cada nó. Já o grupo das arestas, guarda os índices dos nós que a aresta conecta, o peso dessa aresta, bem como o número de inovação dessa aresta.

A mutação pode acrescentar um nó novo, remover um nó, mudar o peso de uma aresta, ou criar uma aresta nova.

Para o caso de criar uma aresta nova, ela simplesmente adiciona um elemento na lista de arestas do DNA com a nova aresta, seu peso e número de inovação (um incremento ao valor global de inovação da mutação anterior).

No caso de uma mutação mudar o peso de uma aresta, isso pode se dar de duas formas: acrescentando um valor pequeno ao peso já existente ou mudando o peso completo e aleatoriamente.

A mutação para acrescentar um nó ocorre escolhendo primeiro uma aresta. Essa aresta é quebrada em duas e o nó é inserido entre elas. Isso é feito dessa forma para garantir que a rede não piore, e a mudança na topologia tenha alguma chance de sobreviver ao longo das gerações.

Finalmente, a mutação pode remover um nó. Para isso, ela simplesmente marca o mó como *disabled* no DNA da rede.

O problema de se misturar redes com topologias possivelmente bastante diferentes é que uma rede que tente inovar e construir uma estrutura nova, tem pouca chance de sobreviver até seus pesos ficarem balanceados para a nova estrutura. Para resolver esse problema, o NEAT divide a população em espécies.

Para dividir a população em espécies, criamos uma função de proximidade entre as redes, que diz o quão próximo duas redes neurais são, baseado em seu DNA. Todas as redes que tiverem proximidade abaixo de um determinado valor, são consideradas da mesma espécie.

Agora, com a separação em espécies, fazemos o processo de seleção de indivíduos para a população intermetiária, apenas entre espécie. Ou seja, uma rede raramente vai ter que competir com uma rede de outra espécie. Isso ajuda as redes a se desenvolverem até estarem prontas para competir com outras espécies.

Além da evolução dentro de uma mesma espécie, também ocorre uma evolução entre espécies, pegando apenas os melhores de cada uma. Isso garante que não estamos guardando espécies de redes que não tem futuro. Quando uma dada espécie tem poucos indivíduos, acabamos com ela.

O número de inovação entra em jogo na hora do crossover. Ele ajuda a alinhar os genes das redes de forma a saber que estruturas são equivalentes e ter um crossover mais performático. Assim, não precisamos ficar analisando as permutações de topologia para realizar o crossover (o que muitos algoritmos antigos faziam).

Resumidamente o NEAT funciona dessa forma, mas ainda existem muitas perguntas que ficam no ar, para respondê-las, basta ler o artigo original.

Além dessa abordagem para a NEAT, pensamos também em evoluir uma rede que avalia o tabuleiro de forma ótima para usar em conjunto com o minimax.

Outra estrategia que pensamos em implementar foi a de aprendizado por reforço. Usaríamos a estrategia de Deep Q-Learning para treinar a rede. Basicamente, essa estrategia utiliza um algoritmo básico de aprendizado por reforço para problemas que seguem uma cadeia de Markov e utiliza deep learning para treinar a função Q (responsável por avaliar cada par, estado-ação). Essa estrategia foi logo abandonada por demandar muito tempo para treinar uma rede desse tipo.

Finalmente, pesnamos em ficar com o minimax, mas explorar opções mais avançadas. Para isso, pesquisamos na internet e encontramos outras otimizações do minimax, como por exemplo o NegaScout. Entretanto, optamos pela estrategia conhecida como MTD-f. De acordo com nossas pesquisas, é a forma de minimax mais eficiente atual.

Resumidamente, o MTD-f percorre uma árvore de minimax múltiplas vezes, sempre buscando zero-windows, ou seja lugares onde o intervalo no corte Alpha-Beta é inexistente ou zero, lugares que poderíamos podar a árvore. A ideia é ir podando a árvore multiplas vezes, sempre alterando um *lowerbound* e um *upperbound* do valor do estados estados que estamos interessados (os da próxima jogada). Quando o *lowerbound* e upperbound convergirem para um memo valor, sabemos que esse é o valor exato daquele estado e podemos tomar a nossa decisão sobre qual estado deveríamos ir.

Chegamos a começar a implementação desse método, entretanto não conseguimos terminar devido a um pequeno problema. Como o MTD-f realiza diversas buscas na árvore, é necessário termos uma tabela de transposição, ou seja, uma tabela com valores que já calculamos, para ser eficiente. Com essa tabela, verificamos primeiro se já temos um valor para determinado nó, antes de explorá-lo. Se tivermos, usamos esse valor, caso o contrário, exploramos o nó.

A questão é que usando o corte Alpha-Beta no minimax, nem sempre temos o valor exato de cada nó. Na verdade, qunado o corte Alpha-Beta está realmente sendo aproveitado, praticamente não temos valores reais de nós. Temos apenas limites superiores e inferiores.

Com isso, acabamos nos confundindo na implementação dessa tabela de transposição e não conseguimos implementá-la corretamente, aleijando, consequentemente, toda a melhoria do MTD-f.

Estrategias de alto nível

Tendo uma visão mais de alto nível sobre o Othello, chegamos a pesquisar que estrategias são melhores. Durante essa pesquisa, nos deparamos com dois grandes grupos de estrategias, que são mais comumente usadas: estrategias posicionais e estrategias de mobilidade.

As estratégias posicionais, são as estrategias que levam em conta apenas a localidade das peças no tabuleiro, por exemplo, como a nossa implementação do minimax usando valores fixos para cada posição no tabuleiro. Nelas, também está inclusa a estratégia de contar o número de peças de cada jogador, que essencialmente seria um tabuleiro no qual todos os pesos são 1.

As estratégias de mobilidade, levam em consideração outros aspectos do jogo, como por exemplo o número de espaços disponíveis para cada jogador jogar. Uma estrategia muito comum de mobilidade é deixar um jogador com poucas possibilidades de jogada, para que o jogador usando a estrategia de mobilidade possa determinar e forçar o outro jogador a jogar em determinadas posições. Outras estrategias de mobilidade levam em conta também as peças que cada jogador tem na fronteira e no miolo do jogo.

É bem sabido que estratégias de mobilidade são extremamente melhores que as posicionais. Se uma for usada contra a outra, em uma execução perfeita, a de mobilidade sempre ganha. E é aí que está o problema: execução perfeita. As estrategias de mobilidade são consideravelmente mais complexas para jogadores humanos aprenderem e também de serem implementadas algoritmicamente.

O minimax jogando com uma heurística de mobilidade sempre ganha de um minimax usando uma heurística posicional.

Fontes de pesquisa:

http://people.csail.mit.edu/plaat/mtdf.html
http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf
http://homepages.cwi.nl/~paulk/theses/Carolus.pdf
https://en.wikipedia.org/wiki/MTD-f
https://en.wikipedia.org/wiki/Principal_variation_search
https://en.wikipedia.org/wiki/Quiescence_search
https://www.nervanasys.com/demystifying-deep-reinforcement-learning/
Entre outros cujos links foram perdidos ao longo do tempo...