

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



**UM AGENTE JOGADOR DE GO COM BUSCA EM ÁRVORE
MONTE-CARLO APRIMORADA POR MEMÓRIA
ESPARSAMENTE DISTRIBUÍDA**

MATHEUS ARAÚJO AGUIAR

Uberlândia - Minas Gerais

2013

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



MATHEUS ARAÚJO AGUIAR

**UM AGENTE JOGADOR DE GO COM BUSCA EM ÁRVORE
MONTE-CARLO APRIMORADA POR MEMÓRIA
ESPARSAMENTE DISTRIBUÍDA**

Dissertação de Mestrado apresentada à Faculdade de Ciência da Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Inteligência Artificial.

Orientadora:

Prof^a. Dr^a. Rita Maria Silva Julia

Uberlândia, Minas Gerais
2013

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Ciência da Computação a aceitação da dissertação intitulada “**Um Agente Jogador de Go com Busca em Árvore Monte-Carlo Aprimorada por Memória Esparsamente Distribuída**” por **Matheus Araújo Aguiar** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 4 de Novembro de 2013

Orientadora:

Prof^a. Dr^a. Rita Maria Silva Julia
Universidade Federal de Uberlândia

Banca Examinadora:

Prof. Dr. Carlos Roberto Lopes
Universidade Federal de Uberlândia

Prof. Dr. Luiz Chaimowicz
Universidade Federal de Minas Gerais

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Data: Novembro de 2013

Autor: **Matheus Araújo Aguiar**
Título: **Um Agente Jogador de Go com Busca em Árvore Monte-Carlo
Aprimorada por Memória Esparsamente Distribuída**
Faculdade: **Faculdade de Ciência da Computação**
Grau: **Mestrado**

Fica garantido à Universidade Federal de Uberlândia o direito de circulação e impressão de cópias deste documento para propósitos exclusivamente acadêmicos, desde que o autor seja devidamente informado.

Autor

O AUTOR RESERVA PARA SI QUALQUER OUTRO DIREITO DE PUBLICAÇÃO DESTE DOCUMENTO, NÃO PODENDO O MESMO SER IMPRESSO OU REPRODUZIDO, SEJA NA TOTALIDADE OU EM PARTES, SEM A PERMISSÃO ESCRITA DO AUTOR.

Dedicatória

*Ao meu pai Paulo, minha mãe Ana Maria e meu irmão Felipe.
A todos professores que tive nesta vida.*

Agradecimentos

Agradeço aos meus pais por sempre garantirem meu bem-estar e oferecerem todas as condições para me desenvolver plenamente como ser humano. Agradeço a minha Orientadora e Professora, Rita, pela paciência e confiança no meu potencial. Agradeço também a Rafaela Cristina, minha namorada, pelo apoio e presença que me auxiliaram a encarar com mais disposição o esforço requerido para concluir esta etapa.

“A força de vontade é a alavanca para o progresso em qualquer plano da existência.”
(Desconhecido)

Resumo

Com mais de 4000 anos de história, o jogo de Go é atualmente um dos mais populares jogos de tabuleiro e representa um grande desafio para a Inteligência Artificial. Apesar de suas regras simples, as técnicas que anteriormente obtiveram sucesso em outros jogos como xadrez e damas não conseguem lidar satisfatoriamente com os padrões e comportamentos complexos que emergem durante uma partida de Go. O presente trabalho implementa o SDM-Go, um agente jogador de Go competitivo que procura reduzir a utilização de supervisão no processo de busca pelo melhor movimento. O SDM-Go emprega o modelo de memória esparsamente distribuída como um recurso adicional à busca em árvore Monte-Carlo utilizada por muitos dos melhores agentes automáticos atuais. Baseado no jogador código-aberto Fuego o uso da memória esparsamente distribuída pelo SDM-Go tem como objetivo ser uma alternativa ao processo fortemente supervisionado utilizado por aquele agente. A busca em árvore Monte-Carlo executada pelo jogador Fuego utiliza um conjunto de heurísticas codificadas por profissionais humanos para guiar as simulações e também avaliar novos nós encontrados na árvore. De maneira distinta, o SDM-Go implementa uma abordagem não supervisionada e independente de domínio, onde o histórico dos valores dos estados de tabuleiros previamente visitados durante a busca são utilizados para avaliar novos estados de tabuleiro (nós da árvore de busca). Desta maneira, o SDM-Go reduz a supervisão do agente Fuego, substituindo as heurísticas deste pela memória esparsamente distribuída que funciona como repositório das informações do histórico de estados de tabuleiro visitados. Assim, as contribuições do SDM-Go consistem em: (1) a utilização de uma memória esparsamente distribuída para substituir a abordagem supervisionada do Fuego para avaliar previamente novos nós encontrados na árvore; (2) a implementação de uma representação de tabuleiro baseada em vetores de bits, para não comprometer o desempenho do sistema em função dos tabuleiros armazenados na memória; (3) a extensão da utilização dos resultados das simulações Monte-Carlo para atualizar os valores dos tabuleiros armazenados na memória. Diferentemente de muitos outros agentes atuais, o uso da memória esparsamente distribuída representa uma abordagem independente de domínio. Os resultados obtidos em torneios contra o conhecido agente código-aberto Fuego mostram que o SDM-Go consegue desempenhar com sucesso a tarefa de prover uma abordagem independente de domínio e não supervisionada para avaliar previamente novos nós encontrados na árvore de busca. Apesar do maior tempo de processamento requerido pela utilização da memória esparsamente distribuída, peça central para o desempenho do agente, o SDM-Go consegue manter um nível de jogo competitivo, principalmente no tabuleiro 9X9.

Palavras chave: jogos; agente para go; memória esparsamente distribuída; busca em árvore monte-carlo; simulações

Abstract

The game of Go is very ancient, with more than 4000 years of history and it is still popular nowadays, representing a big challenge to the Artificial Intelligence. Despite its simple rules, the techniques which obtained success in other games like chess and draughts cannot handle satisfactorily the complex patterns and behaviours that emerge during a match of Go. The present work implements the SDM-Go, a competitive agent for Go that seeks to reduce the usage of supervision in the search process for the best move. The SDM-Go utilizes the sparse distributed memory model as an additional resource to the Monte-Carlo tree search, which is used by many of the best automatic Go players nowadays. Based upon the open-source player Fuego, the use of the sparse distributed by SDM-Go has the purpose of being an alternative to the strong supervised process used by Fuego. The Monte-Carlo tree search executed by agent Fuego uses a set of heuristics codified by human professionals to guide the simulations and also to evaluate new nodes found in the tree. In a different way, SDM-Go implements a non-supervised and domain independent approach, where the history of the values of board states previously visited during the search are used to evaluate new boards (nodes of the search tree). In this way, SDM-Go reduces the supervision of Fuego, substituting its heuristics by the sparse distributed memory, which works as a repository for the information from the history of visited board states. Thus, the contributions of SDM-Go consist of: (1) the utilization of a sparse distributed memory to substitute the supervised approach of Fuego to evaluate new nodes found in the search tree; (2) the implementation of a board state representation based on bit vectors, in order to not compromise the performance of the system due to the boards stored in the memory; (3) the extension of the usage of the Monte-Carlo simulation results to update the values of the board states stored in the memory. Distinctly from many other existing agents, the use of the sparse distributed memory represents an approach independent of domain. The results obtained in tournaments against the well known open-source agent Fuego show that SDM-Go can perform successfully the task of providing a non-supervised and independent of domain approach to evaluate new nodes found in the search tree. Despite the longer runtime required by the use of the sparse distributed memory, the core of the agent performance, SDM-Go can keep a competitive level of play, especially at the 9X9 board.

Keywords: game; agent for go; sparse distributed memory; monte-carlo tree search; simulations

Sumário

Lista de Figuras	xix
1 Introdução	21
1.1 Introdução e Motivação	21
1.2 Objetivos	23
1.3 Estrutura da Dissertação	24
2 Fundamentos Teóricos	25
2.1 Jogo de Go	25
2.2 Processo de Decisão Markoviano	28
2.3 Busca em Árvore Monte-Carlo	29
2.4 Memória Esparsamente Distribuída	31
2.5 Operações Bitwise	34
2.5.1 Operação Negação (<i>NOT</i>)	35
2.5.2 Operação E (<i>AND</i>)	35
2.5.3 Operação Ou (<i>OR</i>)	36
2.5.4 Operação Ou Exclusivo (<i>XOR</i>)	36
2.5.5 Deslocamento (<i>Shift</i>)	36
3 Estado da Arte	39
3.1 Fuego	39
3.1.1 <i>Prior Knowledge</i>	41
3.1.2 Representação de Tabuleiro do Agente Fuego	43
3.2 RLGO	43
3.3 Honte	45
3.3.1 NeuroGo	47
4 SDM-Go	49
4.1 Arquitetura Geral	50
4.1.1 Módulo de Busca em Árvore Monte-Carlo	51
4.1.1.1 Melhorando a Busca em Árvore Monte-Carlo com o Algoritmo UCT (<i>Upper Confidence Bound</i>)	60

4.1.1.2	Uso da Memória Esparsamente Distribuída na Busca do SDM-Go	63
4.1.2	Módulo de Representação em Bits do Tabuleiro (BBR)	64
4.1.2.1	Representação de Tabuleiro em Vetor de Bits	65
4.1.2.2	Conversão do Tabuleiro da Busca para Vetor de Bits	67
4.1.3	Atualização da Representação em Bits	69
4.1.3.1	Cálculo de Similaridade entre Tabuleiros	73
4.1.3.2	Operações <i>bitwise</i>	74
4.1.4	Módulo da Memória Esparsamente Distribuída	81
4.1.4.1	Seleção de Protótipos da Memória	85
4.2	Utilização da Memória Esparsamente Distribuída na Fase de Construção da Árvore da Busca	86
4.2.1	Cálculo da Avaliação Prévia do Nó	87
4.3	Estendendo os Resultados das Simulações durante a Fase de Retropropagação	90
4.3.1	Cálculo da Atualização dos Valores dos Protótipos	91
4.4	Limitações da Utilização da Memória SDM para Avaliar os Nós	92
5	Experimentos e Resultados	95
6	Conclusão e Trabalhos Futuros	101
6.1	Trabalhos Futuros	102
	Referências Bibliográficas	105

Lista de Figuras

2.1	Tabuleiro de Go 9X9.	26
2.2	Liberdades de um grupo.	26
2.3	Exemplo de Nakade.	28
2.4	Ilustração do Esquema Geral da Busca em Árvore Monte-Carlo.	30
2.5	Esquema geral da memória esparsamente distribuída.	33
2.6	Exemplos de <i>shifts</i> para a direita e esquerda.	37
3.1	Principais componentes da plataforma Fuego e suas dependências.	40
3.2	Exemplo de <i>auto atari</i> : devido a peça branca 3 , todo o grupo pode ser capturado com uma peça preta na posição a	42
3.3	Exemplos de formas locais utilizadas pelo RLGO.	45
4.1	Arquitetura Geral do SDM-Go	51
4.2	Árvore de busca para configuração de tabuleiro 5x5.	52
4.3	ConFiguração final de jogo no tabuleiro 5x5.	53
4.4	Pseudocódigo da busca em árvore Monte-Carlo implementada pelo agente SDM-Go.	53
4.5	Pseudocódigo do procedimento da fase de construção da árvore da busca em árvore Monte-Carlo implementada pelo agente SDM-Go.	56
4.6	Pseudocódigo do procedimento de simulação de jogo da busca em árvore Monte-Carlo implementada pelo agente SDM-Go.	57
4.7	Pseudocódigo do procedimento de atualização dos nós percorridos na árvore durante um episódio da busca implementada pelo agente SDM-Go.	59
4.8	Ilustração da busca em árvore Monte-Carlo.	61
4.9	Representação de tabuleiro 3X3 usando um vetor de bits.	66
4.10	Pseudocódigo do procedimento de conversão de tabuleiro usado pela busca MCTS para vetor de bits.	68
4.11	Pseudocódigo do procedimento para adicionar uma peça no tabuleiro representado pelo vetor de bits.	70
4.12	Exemplo do cálculo do bit a ser atualizado após a execução de uma ação.	71

4.13	Pseudocódigo do procedimento para adicionar uma peça no tabuleiro representado pelo vetor de bits.	72
4.14	Distância de Hamming entre dois vetores de bits diferentes.	73
4.15	Pseudocódigo do procedimento que calcula o grau de similaridade entre dois vetores de bits.	74
4.16	Pseudocódigo do procedimento para ligar um bit.	75
4.17	Exemplo do processo para ligar um bit do vetor.	76
4.18	Pseudocódigo do procedimento para desligar um bit.	77
4.19	Exemplo do processo para desligar um bit do vetor.	78
4.20	Pseudocódigo do procedimento para calcular a distância de Hamming. . . .	79
4.21	Ilustração do processo para calcular a distância de Hamming utilizando as operações <i>bitwise XOR</i> e <i>Shift</i>	80
4.22	Modelo da memória SDM implementada pelo agente SDM-Go.	82
4.23	Pseudocódigo do Procedimento que Seleciona os Protótipos Mais Similares. .	85
4.24	Pseudocódigo do procedimento utilizado pelo SDM-Go para calcular a avaliação prévia de um nó.	88
4.25	Pseudocódigo do procedimento para atualizar os valores dos protótipos armazenados na memória SDM.	92
4.26	Exemplo do movimento <i>tesuji Crane's Nest</i>	93

Capítulo 1

Introdução

1.1 Introdução e Motivação

Este trabalho apresenta o jogador de Go chamado SDM-Go. Os jogos oferecem características desejáveis para a pesquisa em diversas áreas, principalmente na investigação de teorias e métodos de Inteligência Artificial. Desde a invenção do primeiro computador programável, tem-se buscado implementar não somente jogos, mas também jogadores automáticos inteligentes que ofereçam desafio e até mesmo superem os melhores jogadores humanos. Em 1950, Claude Shanon, definiu as estratégias básicas de um jogador automático de xadrez [Shannon 1950]. Shanon diferencia dois tipos de estratégias: uso de força bruta para explorar todo o espaço de busca e o uso de conhecimento específico ao domínio para examinar apenas uma parte do espaço das jogadas possíveis [Marsland 1997]. Desde então, melhoramentos e refinamentos nas abordagens de busca, representação e aprendizado de conhecimento aliados a potência cada vez mais explorada do processamento paralelo têm gerado técnicas e jogadores automáticos de nível igual ou superior aos grandes mestres humanos [Levene e Bar-Ilan 2005]. Por detrás da atividade lúdica que usualmente caracteriza os jogos, encontramos as características que os tornam ambientes atrativos para pesquisadores da área de Inteligência Artificial. A existência de regras formais e bem definidas, a interferência e cooperação de um ou mais agentes, a necessidade de planejamento de uma estratégia geral, o dinamismo e adaptatividade requeridos de um bom jogador, bem como o constante aprendizado deste são propriedades inerentes e desafios estimulantes que justificam e propulsionam o interesse e a pesquisa sobre este domínio [Russell e Norvig 2004]. Os jogos se relacionam com diversos problemas encontrados no mundo real, onde as regras são menos formais e menos determinísticas, gerando situações inesperadas e imprevisíveis. Apesar disto, o desenvolvimento de métodos que tenham sucesso nos jogos, leva a soluções satisfatórias no mundo real, que podem não ser ótimas em todos os casos, mas que resolvem a grande maioria das instâncias de um dado problema real. Alguns exemplos destes problemas reais, cuja solução é investigada

através de métodos inteligentes são:

1. a interação entre máquinas e humanos: já é comum o diálogo de seres humanos com máquinas nas mais diversas situações como, por exemplo, os atendentes eletrônicos de empresas de prestação de serviços. O agente ELVIS (*Elvis Voice Interactive System*) [Walker 2000] é um exemplo de programa inteligente que aprende a escolher uma estratégia ótima de diálogo por meio de suas experiências e interações com usuários humanos;
2. controle de tráfego urbano: neste caso, deseja-se criar um agente capaz de controlar o número médio de veículos sobre uma rede urbana de forma a minimizar os congestionamentos e o tempo de viagem [Thorpe e Anderson 1996];
3. navegação autônoma de robôs: a partir de um ponto de referência inicial, um mapa deve ser obtido autonomamente por um robô de maneira que este aprenda uma trajetória que o leve até um ponto alvo, ao mesmo tempo, desviando de obstáculos [Monteiro e Ribeiro 2003].

Assim, a primeira motivação para este trabalho é o fato de que implementar jogadores automáticos de Go envolve dificuldades de ordem teórica e técnica bastante similares àquelas encontradas no tratamento de vários problemas que ocorrem no cotidiano, tal como os apresentados acima. Estas similaridades ocorrem devido ao fato de que, assim como Go, todos estes problemas requerem combinações estratégicas para tomar decisões em situações onde existem outros agentes que interferem nos efeitos positivos das ações de um agente. Além disso, os agentes que atuam nestes domínios necessitam aprender durante sua interação com o ambiente assim como durante as mudanças de estado que ocorrem após a execução de uma ação. A segunda motivação que levou a este trabalho é a grande complexidade que o problema possui, já que o jogo de Go pertence a um conjunto de problemas com um grande estado de espaços e fator de ramificação, como visto na Tabela 1.1 [Campos e Langlois 2003]. O fator de ramificação do jogo de Go fica atrás apenas do correspondente ao jogo de gamão. No tamanho do espaço de estados, supera em muito os demais jogos apresentados. Apesar da simplicidade das regras do jogo de Go, a falta de uma boa heurística dificulta o emprego de técnicas que obtiveram sucesso em xadrez e em outros jogos como damas e gamão. Surge, então, a necessidade de novas abordagens e ideias que possam ser aplicadas com sucesso no jogo Go.

Tabela 1.1: Complexidade de alguns jogos de tabuleiro.

Jogo	Ramificação	Estados
Xadrez	30-40	10^5
Damas	8-10	10^{17}
Gamão	± 420	10^{20}
Go 19x19	± 360	10^{160}

Atualmente, apesar de melhora crescente, os mais avançados jogadores automáticos ainda mantêm um nível de jogo bem inferior ao de profissionais e mestres humanos de Go [Bouzy e Cazenave 2001] [Kocsis e Szepesvari 2006]. Dentre estes melhores jogadores automáticos, é interessante citar os seguintes: Fuego [Enzenberger et al. 2010], GnuGo [FSF 2013], RLGO [Silver et al. 2007] e Crazy Stone [Coulom 2006]. O agente jogador Fuego é implementado sobre uma plataforma de mesmo nome, cujo código é aberto e disponível gratuitamente. Esta plataforma tem sido amplamente utilizada como ambiente para subsidiar a implementação de outros agentes de Go [Enzenberger et al. 2010]. No jogador Fuego, a busca pelo melhor movimento é desempenhada pelo algoritmo de Busca em Árvore Monte-Carlo (*Monte-Carlo Tree Search* - MCTS) [Coulom 2006] [Paris et al. 2003] [Gelly et al. 2006] [Browne et al. 2012]. O agente Fuego emprega na busca várias heurísticas específicas do jogo de Go, o que caracteriza sua maneira fortemente supervisionada de operar.

O SDM-Go propõe-se a reduzir a supervisão do agente Fuego. Através da utilização da memória SDM, o agente aproveita o conhecimento histórico obtido de execuções anteriores do processo busca, diferentemente do Fuego.

Os resultados obtidos em torneios contra o bem conhecido agente código aberto Fuego confirmam os benefícios oferecidos pelas melhorias que são introduzidas no SDM-Go. Estas melhorias são devido ao aumento da precisão das avaliações dos nós obtidas com o uso da memória SDM. Tais resultados propiciaram o aceite para publicação de artigo completo na conferência internacional referenciada em [Aguilar e Julia 2013].

1.2 Objetivos

O agente SDM-Go proposto neste trabalho busca o aprimoramento do agente Fuego, reduzindo a supervisão através da inserção de um processo de aprendizado não supervisionado. Os objetivos principais do SDM-Go são:

1. estender a utilização dos resultados das simulações da busca executada pelo agente Fuego de maneira a aumentar a precisão das avaliações dos nós da árvore;
2. reduzir a supervisão do Fuego através da substituição da estratégia *Prior Knowledge*.

Para alcançar estes objetivos, o SDM-Go melhora a busca MCTS do Fuego através do uso da memória esparsamente distribuída (*Sparse Distributed Memory* - SDM) [Kanerva 1988]. Este modelo de memória funciona a partir do casamento parcial de padrões de entrada com os padrões armazenados. No começo do jogo, a memória está vazia e os valores iniciais dos estados do tabuleiro são todos 0, que é um dos fatores que distinguem o SDM-Go do Fuego, uma vez que este último utiliza *Prior Knowledge* para iniciar estes valores (conforme será apresentado, o presente trabalho, a título comparativo, testará o

desempenho do Fuego jogando, ora com, ora sem *Prior Knowledge*). Neste contexto, a construção da árvore de busca pelo SDM-Go também é guiada pelas avaliações UCB e pelas simulações. Entretanto, o agente estende os resultados das simulações nas seguintes maneiras:

1. os nós simulados que diferem em **pelo menos** 30% (em termos da distância de Hamming [Spinelli 2003]) de outros nós da memória SDM são armazenados na memória, juntamente com seus respectivos valores;
2. todos os nós envolvidos na construção da árvore Monte-Carlo (MC) que são **pelo menos** 70% similares (também em termos da distância de Hamming) com algum estado S armazenado na memória SDM têm seus valores atualizados (ponderados de acordo com o grau de similaridade em relação a S);
3. o resultado de cada simulação é usado para atualizar não somente os valores dos nós da árvore MC que pertencem ao caminho daquela simulação, assim como o faz o Fuego, mas também os valores dos nós armazenados na memória SDM que são **pelo menos** 70% similares a algum nó do caminho.

1.3 Estrutura da Dissertação

O restante deste trabalho está organizado em capítulos, conforme disposto a seguir:

- **Capítulo 2**, onde o referencial teórico das técnicas empregadas na construção do SDM-Go é apresentado: busca em árvore Monte-Carlo, memória esparsamente distribuída e operações *bitwise*, dentre as mais importantes.
- **Capítulo 3**, que apresenta o estado da arte dos programas jogadores de Go que utilizam técnicas de inteligência artificial para aprender e realizar boas jogadas.
- **Capítulo 4**, onde a implementação do SDM-Go, um agente jogador de Go com busca MCTS aprimorada por memória SDM, é descrita em maiores detalhes. Além disso, os resultados dos testes realizados com o SDM-Go são apresentados e discutidos.
- **Capítulo 5**, finaliza com as conclusões e possíveis linhas de investigação futura a partir da implementação e resultados obtidos com este trabalho.

Capítulo 2

Fundamentos Teóricos

Este capítulo apresenta o referencial teórico utilizado na implementação do SDM-Go.

2.1 Jogo de Go

A história do jogo de Go conta com aproximadamente 4000 anos [Yang e Turner 2005] e o jogo ainda desfruta de bastante popularidade no mundo. Apesar das regras simples, a complexidade do Go tem frustrado as muitas tentativas de construir um bom agente jogador desde a década de 70 [Bouzy e Cazenave 2001] [Gelly e Wang 2006]. Go é um jogo de tabuleiro para dois jogadores que alternadamente posicionam uma peça por vez da sua respectiva cor, branca ou preta. Um tabuleiro de Go $N \times N$, com N comumente variando de 9 até 19, é formado por N linhas horizontais e N linhas verticais, conforme ilustrado na Figura 2.1.

No tabuleiro da Figura, as peças são posicionadas sobre as intersecções das linhas que formam o tabuleiro. Os pontos A até F marcados na Figura são exemplos de intersecções onde peças já foram posicionadas ou ainda podem ser posicionadas. Conforme será explicado posteriormente, é possível existir um ponto vazio do tabuleiro onde nenhuma peça pode ser posicionada. O objetivo principal do jogo é assegurar com grupos de peças a maior parte do tabuleiro. Um princípio básico do jogo de Go é que as peças necessitam ter pelo menos uma liberdade para permanecer no tabuleiro. Uma **liberdade** é um ponto vazio adjacente à peça. A Figura 2.2 ilustra uma peça preta e suas 4 liberdades. Ao longo da sequência mostrada na Figura, de A até D as liberdades da peça preta são removidas até restar apenas uma. Nesta situação, é dito que a peça está em **atari**, ou seja, que está na iminência de ser capturada, bastando apenas um movimento que retire sua última liberdade.

Um **grupo** é um conjunto de peças de mesma cor adjacentes horizontal ou verticalmente, como, por exemplo, o grupo de peças brancas que ocupa o ponto D da Figura 2.1. Peças adjacentes diagonalmente não são consideradas conectadas. As peças de um mesmo grupo compartilham suas liberdades e um grupo é capturado quando todas suas

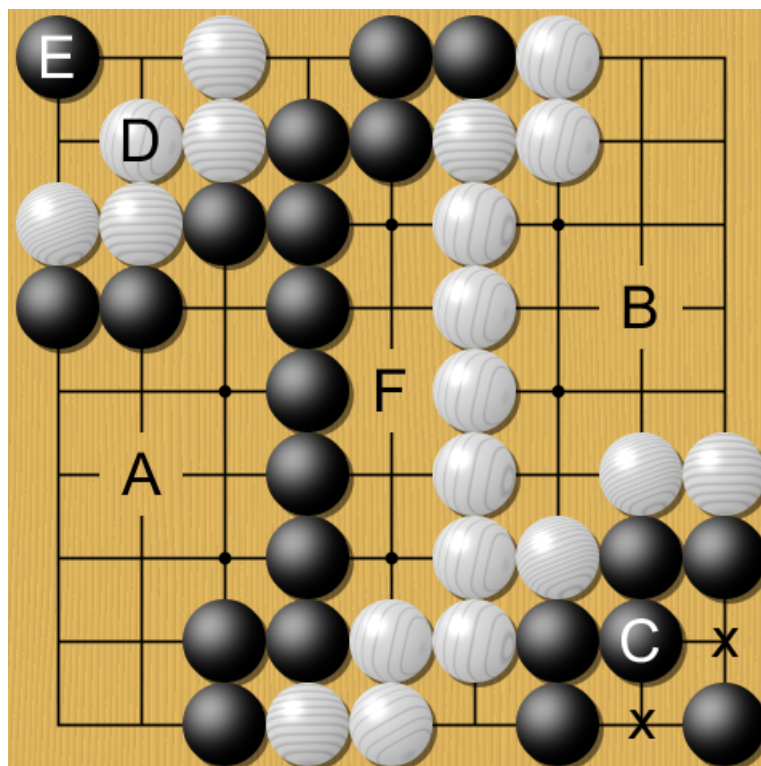


Figura 2.1: Tabuleiro de Go 9X9.

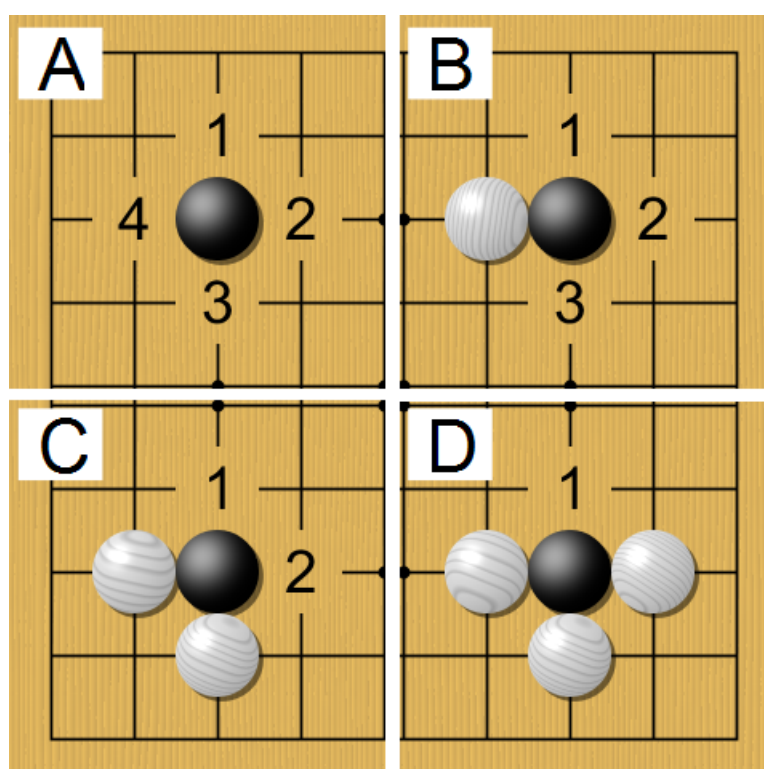


Figura 2.2: Liberdades de um grupo.

liberdades são retiradas. Um ponto vazio do tabuleiro que esteja cercado de peças de um mesmo grupo, é chamado de **olho**, como, por exemplo os pontos marcados com **x** na Figura 2.1. O oponente não consegue colocar uma peça em nenhum destes pontos, pois,

de imediato, ela não possuiria nenhuma liberdade e não poderia, portanto, permanecer no tabuleiro. Para garantir que um grupo nunca seja capturado são necessários dois olhos, quando então é dito que o grupo está **vivo** incondicionalmente. Existem situações mais complexas onde outros fatores determinam se o grupo será ultimamente capturado ou não. E há ainda situações chamadas de vida mútua, onde a vida de um grupo depende da vida do grupo inimigo e vice-versa. Grupos que não possuem dois olhos ou não tem espaço suficiente para formar dois olhos são chamados de mortos, pois não podem resistir a uma tentativa de captura. Na Figura 2.1 o grupo de peças pretas que ocupa o ponto C está vivo, enquanto o grupo de peças brancas que ocupa o ponto D está morto. Uma peça ou grupo de peças é retirado do tabuleiro quando capturado. No jogo de Go, considera-se que o jogador das peças pretas, quem começa a partida posicionando a primeira peça, tem vantagem sobre o jogador das peças brancas. Para equilibrar a situação, é comum dar ao jogador das peças brancas uma pontuação adicional chamada **komi**, com valor que varia, geralmente, entre 5.5 e 7.5. A fração 0.5 é utilizada para decidir empates a favor das peças brancas. Ao final do jogo, a pontuação de cada jogador é dada pelo número de pontos do tabuleiro que estão totalmente cercados por peças de sua respectiva cor. O jogador com o maior número de pontos cercados, ou seja, o maior território, considerando ainda o *komi*, é o vencedor. A estratégia geral durante o jogo consiste em ocupar o máximo de território, garantir possibilidades de formar grupos com dois olhos e tentar impedir o oponente de fazer o mesmo. O conceito de forma é muito importante no jogo de Go e é empregado para utilizar as peças de maneira eficiente, atingindo estes objetivos básicos da estratégia geral. Várias heurísticas são utilizadas para obter uma boa forma, ou configuração de peças, e estão intimamente ligadas com a questão de vida ou morte dos grupos. O movimento *Nakade* é um exemplo de heurística que identifica a posição vital de uma dada situação. Conforme ilustrado na Figura 2.3, o ponto vital da forma do grupo de peças brancas é aquele marcado por A. Se uma peça branca é posicionada em A, o grupo forma dois olhos e está vivo incondicionalmente. Porém, se uma peça preta é posicionada ali, o grupo de peças brancas apenas consegue formar um olho e é considerado morto, pois pode ser capturado.

Várias outras heurísticas são baseadas em situações específicas do jogo de Go. Utilizando apenas de conceitos já apresentados, um outro exemplo de heurística diz respeito a encontrar o movimento que coloca um grupo inimigo em atari. Ou ainda, encontrar o movimento que previne tal situação.

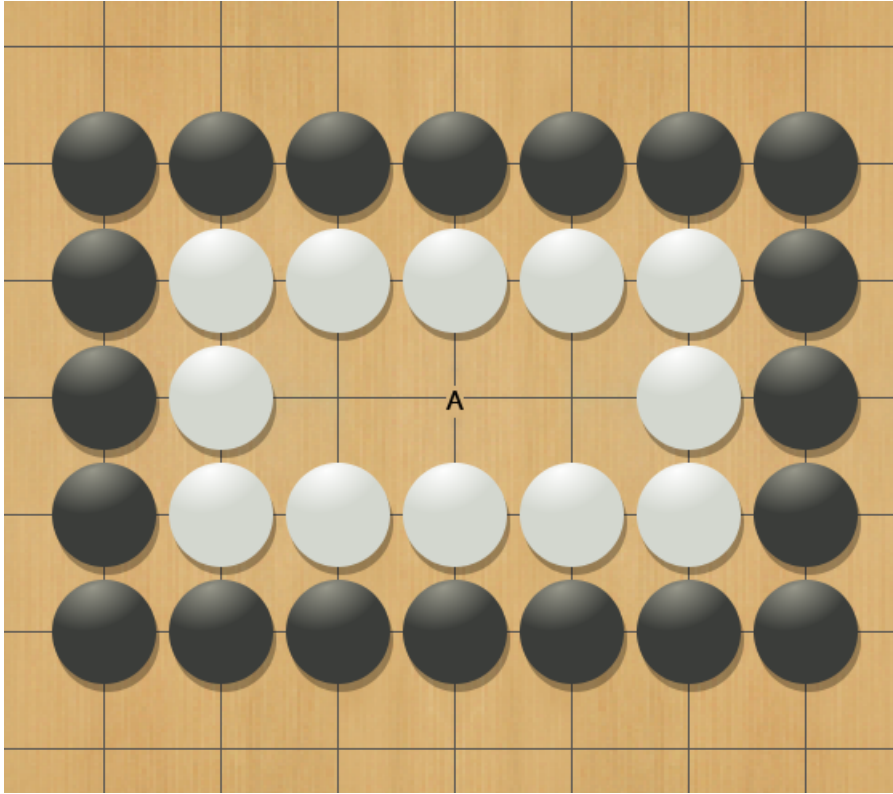


Figura 2.3: Exemplo de Nakade.

2.2 Processo de Decisão Markoviano

Um processo de decisão markoviano consiste de um conjunto de estados S e um conjunto de ações A , onde existe uma função de transição que indica a probabilidade de executar uma ação disponível $a \in A$ a partir de um estado $s \in S$ e terminar num estado $s' \in S$. Tal função pode ser denotada como $P(s, a, s') = \Pr(s_{t+1} = s' | s_t = s, a_t = a)$. Define-se também uma função de recompensa, especificando o reforço recebido em uma dada transição de estados, denotada como $R(s, a, s') = E[r_{t+1} | s_t = s, a_t = a, s_{t+1} = s']$, onde E significa esperança estatística. O procedimento para definir a escolha de uma ação a em um estado s , é chamado de uma política $\pi(s, a) = \Pr(a_t = a | s_t = s)$. Uma política π faz o mapeamento da distribuição de probabilidade de ocorrência sobre todas as ações disponíveis em s . A função de avaliação $V_\pi(s)$ retorna a recompensa acumulada esperada ao seguir a política π a partir de s . Similarmente, uma função de avaliação de ação $Q_\pi(s, a)$ é a recompensa acumulada esperada ao tomar a ação a no estado s e depois seguir a política π . Jogos de dois jogadores, informação perfeita e soma-zero (*zero-sum*) como xadrez, damas e Go, podem ser vistos como uma generalização de um processo de decisão markoviano [Littman 1994] [Silver e Müller 2012]. Tal generalização, usualmente chamada de jogo simétrico alternante markoviano, é formada por um agente e um oponente que alternadamente escolhem ações de um mesmo conjunto A , seguindo políticas diferentes π_1 e π_2 . Uma política *self-play* é tal que $\pi = \pi_1 = \pi_2$, ou seja, tanto agente quanto oponente utilizam a mesma distribuição de probabilidade π para selecionar suas

ações. O agente tenta sempre maximizar sua recompensa acumulada, enquanto o oponente procura minimizar a recompensa total do agente. Uma política *minimax* ótima é uma política *self-play* que maximiza a função de avaliação do agente e minimiza a função de avaliação do oponente, correspondendo a jogadas ótimas na suposição de que o oponente jogue perfeitamente. O jogador SDM-Go implementado neste trabalho é concebido considerando o jogo Go como um processo de decisão markoviana.

2.3 Busca em Árvore Monte-Carlo

A busca em árvore Monte-Carlo (MCTS) consiste na simulação de um número arbitrário de episódios para construir uma árvore de busca [Coulom 2006] [Chaslot et al. 2006]. Cada episódio corresponde a uma simulação de uma partida possível do jogo de Go, a partir do estado corrente. Os episódios são constituídos de quatro passos característicos do método Monte-Carlo de busca em árvore:

1. **seleção:** neste passo, um caminho será percorrido, a partir do estado corrente, até encontrar um nó cujos filhos ainda não pertencem à árvore. A escolha dos nós que fazem parte deste caminho é orientada por uma política de seleção $\pi_{selecao}$, baseada na qualidade dos nós, na frequência com que foram visitados durante as simulações e na frequência com que levaram à vitória no final de cada simulação;
2. **expansão:** a árvore aumenta de tamanho com a inclusão do filho com melhor avaliação do nó folha do caminho definido pela seleção;
3. **playout:** partindo do nó recém adicionado à árvore, uma política *self-play* $\pi_{simulacao}$ conduz as jogadas até uma situação de final de jogo. Tal política pode ser baseada em regras heurísticas que apontam para o tipo de movimento que deve ser executado a partir de cada nó envolvido no *playout*. É importante ressaltar que nenhum nó visitado durante o *playout* é adicionado à árvore de busca;
4. **retropropagação:** o estado final do jogo é avaliado e o resultado do final do jogo é retropropagado pelo caminho percorrido na árvore no episódio corrente. Os valores associados aos nós deste caminho são atualizados de acordo com tal resultado. Aqui termina um episódio de simulação de uma partida.

Os dois primeiros passos da busca, **seleção** e **expansão**, serão também referenciados neste trabalho como a fase de **construção da árvore**. Desta maneira, a árvore de busca é construída iterativamente, ao mesmo tempo que os valores estimados de seus nós são progressivamente atualizados no decorrer dos episódios, de modo que a acuidade deles aumenta na medida em que ocorrem mais simulações. Os quatro passos da busca MCTS descritos anteriormente são ilustrados na Figura 2.4.

Uma das versões mais utilizadas da busca em árvore Monte-Carlo é o algoritmo UCT (*Upper Confidence Bound*). Este algoritmo é utilizado para escolher os nós durante a fase

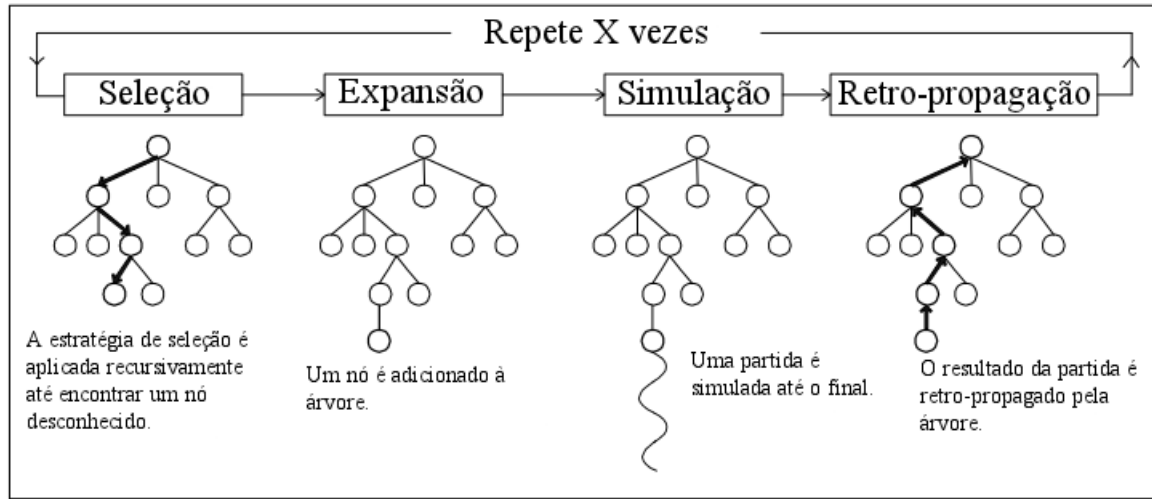


Figura 2.4: Ilustração do Esquema Geral da Busca em Árvore Monte-Carlo.

de construção da árvore, isto é, ele corresponde à política $\pi_{selecao}$. Durante as simulações, cada estado de tabuleiro é representado por um nó s na árvore. Cada nó possui pelo menos as seguintes informações armazenadas:

1. $N^t(s)$, que representa o número de simulações nas quais o estado s foi visitado até o tempo t ;
2. $Q^t(s,a)$, o valor estimado de escolher a ação a a partir de s no instante t ;
3. $N^t(s,a)$, o número de vezes em que a ação a foi executada a partir de s , durante as simulações ocorridas até o tempo t .

O valor $Q(s,a)$ representa, de fato, a avaliação estimada do nó s_i , que é o filho de s obtido apela simulação da ação a a partir de s . Tal valor corresponde à média de vitórias obtidas em *playouts* cujo caminho passa por s_i na busca corrente. Neste trabalho, tal parâmetro será chamado **qualidade** do nó s_i . A atualização dos valores dos nós da árvore MC, a partir do resultado R de um dado *playout*, é feita através das equações 2.1, 2.2, 2.3 mostradas a seguir:

$$N^{t+1}(s) = N^t(s) + 1 \quad (2.1)$$

$$N^{t+1}(s, a) = N^t(s, a) + 1 \quad (2.2)$$

$$Q^{t+1}(s, a) = Q^t(s, a) + \left(R - \frac{Q^t(s, a)}{N^{t+1}(s, a)}\right) \quad (2.3)$$

A política UCT, no processo de escolha dos nós durante a construção da árvore, procura balancear a exploração entre nós pouco simulados e nós cujos valores estimados são mais altos. Assim, a escolha dos nós não é dominada apenas pelo termo de qualidade

$Q^t(s, a)$. A avaliação de um nó s na política UCT considera também a frequência de visitas a s durante as simulações. Como o valor do termo $Q^t(s, a)$ é dependente, essencialmente, da qualidade e do número das simulações previamente realizadas, o algoritmo UCT emprega um segundo termo que privilegia os nós com baixa frequência de visitas. Assim, a avaliação $V_{UCT}(s, a)$ de um nó s_i gerado ao executar a ação a a partir de s no tempo t é dada pela equação 2.4 [Auer e Fischer 2002], apresentada a seguir:

$$V^t(s, a) = Q^t(s, a) + C * \sqrt{\frac{2 \log N^t(s)}{N^t(s, a)}} \quad (2.4)$$

O segundo termo da equação, que é somado ao termo de **qualidade**, representa o número de vezes em que a ação a foi escolhida quando uma simulação visitou s . Ou seja, este termo é a **frequência** de visitas ao nó s_i durante os episódios de simulação da busca. C é uma constante arbitrária que controla o balanceamento na equação entre a **qualidade** e **frequência** de um nó. Sempre que um nó s_i possui valor nulo de **frequência**, ou seja, nunca foi visitado, o algoritmo UCT prefere s_i sobre outros nós que já foram visitados. Assim, durante a fase de construção da árvore, o movimento a^* escolhido é aquele que maximiza a equação 2.5:

$$a^* = \operatorname{argmax} V(s, a) \quad (2.5)$$

Conforme apresentado na Seção 4.1.1, o presente trabalho SDM-Go concentra seu esforço de aprimoramento do desempenho do agente justamente modificando o tratamento dado ao parâmetro **qualidade** em relação ao jogador Fuego.

2.4 Memória Esparsamente Distribuída

Inspirado pela memória humana, Petti Kanerva desenvolveu um modelo matemático [Kanerva 1988] que procura relacionar novos padrões de entrada com padrões previamente conhecidos, de acordo com a similaridade entre eles. A principal ideia que Kanerva buscava ao desenvolver este modelo de memória é simples: as distâncias entre conceitos na mente humana correspondem às distâncias entre pontos de um espaço hiper-dimensional (o número de dimensões de tal espaço é relativamente grande, de pelo menos centenas de dimensões). Um conceito que é representado por um vetor neste espaço hiper-dimensional não necessariamente precisa ser exatamente codificado. Isto acontece devido a distribuição de pontos num espaço hiper-dimensional. Dado um ponto de interesse qualquer, ele está localizado bastante distante do restante dos outros pontos do espaço. Desta maneira, a representação pode ser fracamente aproximada antes que um ponto seja confundido com outro. Neste sentido, os vetores deste espaço hiper-dimensional são robustos ou tolerantes à falhas e um dispositivo baseado neles pode aproveitar esta robustez [Kanerva

1988]. Assim, o modelo de memória esparsamente distribuída (SDM) funciona através de casamentos parciais entre as entradas apresentadas e os padrões previamente armazenados. A informação é codificada em longos vetores de bits, com centenas ou milhares de posições. Usualmente, a medida de similaridade utilizada para relacionar os vetores de bits é a distância de Hamming [Spinelli 2003], que mede o número de bits nos quais dois vetores diferem. Por exemplo, a distância de Hamming entre **11001** e **01101** é 2. Outra propriedade dos espaços hiper-dimensionais está relacionada com a distância entre os pontos. Em média, dois pontos que sejam escolhidos aleatoriamente estarão relativamente distantes um do outro. Dificilmente, então, estes dois pontos estarão relacionados. Entretanto, entre eles existem muitos outros pontos que são próximos a ambos. A exceção do tamanho dos endereços e dados, a memória SDM é bastante semelhante a uma memória de acesso aleatório (RAM) comum utilizada pelos computadores atuais. Conforme mostrado na Figura 2.5 [Denning 1989], a memória consiste de 3 registradores especiais que armazenam o **endereço** a ser acessado, a informação a ser armazenada (**entrada**) e a informação a ser recuperada (**saída**). Além disto, a memória conta também com um vetor de endereços, cada um apontando para um local de memória correspondente. No exemplo do modelo apresentado na Figura 2.5, os endereços e padrões de entrada e saída são sequências binárias e os locais de memória são vetores de inteiros. Diferentemente de uma memória RAM comum, onde apenas uma localidade é acessada por operação, neste modelo de memória, mais de um local pode ser selecionado. Para isso, a memória SDM possui um vetor de seleção que indica quais as entradas são utilizadas durante as operações de leitura e escrita. As entradas da memória SDM são selecionadas de acordo com o grau de similaridade entre o endereço fornecido no registrador e os endereços armazenados no vetor. O grau de similaridade limite que define a seleção de um endereço de memória é chamado de **raio de seleção** e geralmente é um valor arbitrário determinado previamente. Desta maneira são selecionadas apenas as entradas da memória cujo grau de similaridade com o endereço fornecido no registrador atende ao critério do raio de seleção. Tais entradas são indicadas no vetor de seleção, conforme mostrado na Figura 2.5 pelas posições sombreadas deste vetor e são, então, submetidas a uma operação de entrada/saída desejada. Caso seja uma operação de escrita, a informação contida no registrador de entrada será armazenada na memória. No exemplo da Figura, para cada bit 1 da entrada, o inteiro na posição correspondente de cada local de memória selecionado será incrementado por 1 e para cada bit 0, decrementado por 1. Se a operação desejada é de leitura, a informação contida nos locais de memória selecionados será processada e o resultado colocado no registrador de saída. No exemplo da Figura, os valores dos inteiros de mesma posição dos locais de memória selecionados são somados e depois truncados. Se o valor n da soma em determinada posição é maior que 0, ele é truncado para 1. Em caso contrário, se n é menor ou igual a 0, ele é truncado para 0.

O modelo de memória esparsamente distribuída pode ser implementado de várias

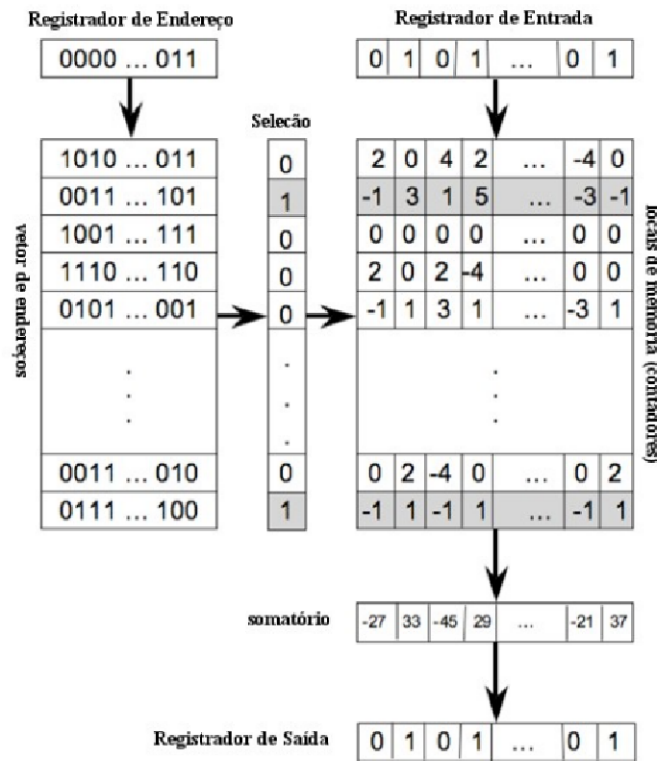


Figura 2.5: Esquema geral da memória esparsamente distribuída.

maneiras, como uma memória associativa comum ou como uma rede neural. Em problemas de aprendizado por reforço a memória SDM pode ser empregada como uma forma de representar os estados. Geralmente, nestes casos, ela também é chamada de *Kanerva Coding* [Sutton e Barto 1998]. Para reduzir a complexidade da função de avaliação, bem como a memória necessária para armazenar uma Tabela de estados, a memória SDM é carregada (prévia ou dinamicamente) com um conjunto de protótipos. Tais **protótipos** representam situações ou estados do problema, de maneira que para melhores desempenhos a memória seja iniciada, no caso de ser previamente carregada, ou retenha, no caso de dinamicamente construída, aqueles protótipos que carreguem mais informações pertinentes ao aprendizado. Então, cada protótipo do conjunto armazenado na memória é comparado com os estados do problema que aparecem durante a experiência real do agente de aprendizado. Define-se $\mu(j, s)$, o grau de adjacência entre um protótipo P_j e o estado s , igual a 1 quando o grau de similaridade entre s e P_j atende ao limite estabelecido pelo raio de seleção. Em caso contrário, o valor do grau de adjacência $\mu(j, s)$ é igual a 0. Alternativamente, conforme explicado mais adiante, também é possível considerar o próprio grau de similaridade como grau de adjacência. Cada protótipo P_j possui um valor θ_j associado e uma aproximação da avaliação do estado s é formada pela soma dos valores

θ_j dos protótipos adjacentes, conforme apresentado pela equação 2.6 a seguir:

$$V(s) = \sum_{j=0}^K \theta_j * \mu(j, s) \quad (2.6)$$

onde $V(s)$ é a aproximação obtida através do conjunto de protótipos similares ao estado s e K é o tamanho total do conjunto de protótipos armazenados na memória. Desta maneira, apenas os valores dos protótipos cujo grau de similaridade é 1 são utilizados no cálculo do valor aproximado de s . A aproximação da função de avaliação é melhor quando cada estado s é adjacente a um único conjunto de protótipos. É possível que os protótipos escolhidos, que formam a representação do estado, não estejam bem distribuídos pelo espaço de estados do problema. Quando tal condição ocorre, muitos estados podem não ser adjacentes a nenhum protótipo ou ainda adjacentes a um mesmo conjunto. Estas colisões e carências de protótipos degradam a qualidade de um agente que utiliza o *Kanerva Coding*, pois reduzem sua capacidade de diferenciar e avaliar estados distintos. Algumas otimizações são propostas em [Wu e Meleis 2009] para contornar este problema e basicamente consistem em, após a memória ser iniciada aleatoriamente, inserir novos protótipos enquanto outros são descartados. Uma abordagem semelhante é utilizada pelo agente SDM-Go, implementado neste trabalho, conforme será visto na Seção 4.2.1. Outra otimização proposta em [Wu e Meleis 2009], consiste em utilizar um grau de adjacência *fuzzy* $\mu_{fuzzy}(j, s)$ que é calculado em função da distância de Hamming entre o estado s e o protótipo P_j . Desta maneira, o grau de adjacência $\mu_{fuzzy}(j, s)$ assume o valor 1 quando o estado s for idêntico ao protótipo P_j . Em caso contrário, quando s é completamente diferente de P_j , $\mu_{fuzzy}(j, s)$ assume o valor 0. Além disso, o grau de adjacência $\mu_{fuzzy}(j, s)$ pode também assumir os valores no intervalo contínuo entre 0 e 1, quando s não é nem idêntico, nem totalmente diferente de P_j . Assim, o impacto de um protótipo P_j na avaliação de um estado s do problema é maior quando P_j é mais próximo de s e menor naqueles mais distantes. Estas otimizações produzem uma melhora significativa nos agentes. Propostas similares documentadas em [Hely e Hayes 1999] e [Caraig 2012] também apresentam resultados positivos da utilização da memória SDM. O SDM-Go emprega um método bastante semelhante, também utilizando a distância de Hamming como ponderador no cálculo das avaliações dos nós da árvore de busca, conforme explicado em maiores detalhes na Seção 4.2.1.

2.5 Operações Bitwise

As operações *bitwise* são primitivas que envolvem um ou mais bits implementadas diretamente no processador e que, portanto, são de execução extremamente rápida. Estas operações utilizam a álgebra booleana para transformar os valores dos bits envolvidos. O valor **1** de um bit corresponde ao valor booleano **Verdadeiro**, enquanto o valor **0**

corresponde ao valor booleano **Falso**. As próximas subseções tratam especificamente das operações que são utilizadas pelo agente SDM-Go implementado neste trabalho, conforme apresentado na Seção 4.1.2.

2.5.1 Operação Negação (*NOT*)

A operação de **negação**, também referenciada como **NOT**, é uma das mais simples de todas. Seu funcionamento consiste em inverter o valor de um bit. A operação *NOT* aplicada sobre um bit *b1* tem seu valor armazenado no bit *b2*, conforme mostra a equação 2.5.1 a seguir:

$$b2 = \neg b1 \quad (2.7)$$

A Tabela verdade 2.1 da operação *NOT*, que contém apenas duas entradas, é apresentada a seguir.

Tabela 2.1: Tabela verdade da operação *NOT*.

b1	b2 = $\neg$ b1
0	1
1	0

2.5.2 Operação E (*AND*)

A operação **E** é também referenciada como **AND** e é aplicada sobre dois bits *b1* e *b2*, com o resultado armazenado no bit *b3*, conforme mostrado pela equação 2.8 a seguir:

$$b3 = b1 \wedge b2 \quad (2.8)$$

O valor resultante da operação *AND*, armazenado em *b3*, é 1 se, e somente se, os valores de *b1* e *b2* são ambos 1 também. Em todos os outros casos, o resultado é 0, conforme verificado pela Tabela verdade 2.2 apresentada a seguir.

Tabela 2.2: Tabela verdade da operação *AND*.

b1	b2	b3 = b1$\wedge$b2
0	0	0
0	1	0
1	0	0
1	1	1

2.5.3 Operação Ou (*OR*)

A operação **Ou**, também referida por **OR**, é executada sobre os valores de dois bits $b1$ e $b2$ com o resultado armazenado em $b3$, conforme mostra a equação 2.9 a seguir:

$$b3 = b1 \vee b2 \quad (2.9)$$

O resultado da operação *OR*, armazenado em $b3$, tem valor 0 se, e somente se, ambos os valores de $b1$ e $b2$ também são 0. Nos demais casos, quando pelo menos um dos valores é 1, o resultado é 1. A Tabela verdade 2.3 da operação é mostrada a seguir.

Tabela 2.3: Tabela verdade da operação *OR*.

b1	b2	b3 = b1∨b2
0	0	0
0	1	1
1	0	1
1	1	1

2.5.4 Operação Ou Exclusivo (*XOR*)

A operação lógica **Ou Exclusivo**, abreviada pela sigla **XOR**, é aplicada sobre dois bits $b1$ e $b2$ com o resultado armazenado em outro bit $b3$, escrita conforme a fórmula 2.10 a seguir.

$$b3 = b1 \oplus b2 \quad (2.10)$$

O valor armazenado em $b3$, retornado da operação *XOR*, é 1 se, e somente se, os dois operandos $b1$ e $b2$ possuem valores distintos. Em caso contrário, quando os valores de $b1$ e $b2$ são idênticos, o valor 0 é retornado para $b3$. A Tabela 2.4 apresenta todas as combinações de valores de $b1$ e $b2$ e os respectivos resultados da operação *XOR* armazenados em $b3$.

Tabela 2.4: Tabela verdade da operação *XOR*.

b1	b2	b3=b1⊕b2
0	0	0
0	1	1
1	0	1
1	1	0

2.5.5 Deslocamento (*Shift*)

A operação de **deslocamento**, comumente chamada pelo seu nome em inglês **shift**, tem o papel de transladar os valores dos bits N posições à direita ou à esquerda. O

deslocamento é aplicado sobre todos os bits de um byte. O resultado do *shift* para a direita de N bits do byte $B1$ é armazenado no byte $B2$, conforme mostra a fórmula 2.11 a seguir.

$$B2 = B1 \gg N \quad (2.11)$$

Quando o *shift* é para a esquerda ocorre da mesma maneira, a exceção do operador diferente, conforme apresentado na fórmula 2.12 a seguir.

$$B2 = B1 \ll N \quad (2.12)$$

Durante a operação de deslocamento os valores dos bits são movidos para a esquerda ou para direita, ocupando as posições de seus vizinhos, de acordo com direção do deslocamento. Para entender o *shift* basta imaginar que a sequência de bits é empurrada, ou para a esquerda ou para a direita, até que os valores se desloquem em uma posição. Neste trabalho, considera-se que o bit menos significativo está posicionado sempre mais à direita. Assim, a partir do bit menos significativo, se o deslocamento é para a direita, os valores são substituídos por aqueles do vizinho à esquerda. Caso o bit não possua um vizinho à esquerda, seu valor é 0. De maneira oposta, quando o *shift* é para a esquerda, os valores são substituídos por aqueles do vizinho à direita e no caso de tal vizinho não existir, o valor atribuído ao bit é zero. Uma vez entendido este raciocínio, basta aplicá-lo quantas vezes forem necessárias, para atingir o valor do parâmetro N da operação. A Figura 2.6 a seguir apresenta exemplos de deslocamentos para a esquerda e para a direita. No exemplo da Figura de deslocamento para a direita, os valores dos N bits menos significativos são sempre descartados, enquanto que os N bits mais significativos recebem todos o valor 0. Por outro lado, no exemplo de deslocamento para a esquerda ocorre o contrário, com os valores dos N bits mais significativos sendo descartados, enquanto que os N bits menos significativos recebem todos o valor 0.

Byte Original	Shift para Esquerda	Shift para Direita
7 6 5 4 3 2 1 0 1 0 0 1 0 1 0 1		
N = 1	7 6 5 4 3 2 1 0 0 0 1 0 1 0 1 0	7 6 5 4 3 2 1 0 0 1 0 0 1 0 1 0
N = 2	0 1 0 1 0 1 0 0	0 0 1 0 0 1 0 1
N = 4	0 1 0 1 0 0 0 0	0 0 0 0 1 0 0 1

Figura 2.6: Exemplos de *shifts* para a direita e esquerda.

Neste trabalho, a operação de *shift* é realizada para mover o valor conhecido de um bit para uma dada posição de interesse dentro de um byte. Além disso, o *shift* auxilia

no processo de cálculo da distância de Hamming, quando o valor de cada bit é obtido individualmente, conforme discutido em detalhes na Seção 4.1.3.2.

Capítulo 3

Estado da Arte

Devido a grande variabilidade de abordagens envolvendo a implementação de jogadores automáticos de Go, este capítulo apresenta alguns dos aspectos mais importantes das técnicas do estado da arte. Os trabalhos de outros pesquisadores que são correlacionados com o SDM-Go são descritos e analisados neste capítulo. Tais trabalhos foram importantes ao servirem de inspiração e base teórica para fundamentar a linha de pesquisa e implementação deste trabalho.

3.1 Fuego

A plataforma Fuego é um conjunto de bibliotecas para desenvolver jogadores automáticos para jogos de tabuleiro de dois jogadores, com maior foco no jogo de Go [Enzenberger et al. 2010]. A plataforma Fuego é construída a partir de dois projetos anteriores, o Smart Game Board [Kierulf 1990] e o Explorer [Müller 2002]. O primeiro reúne uma coleção de ferramentas para jogos de dois jogadores, enquanto o segundo é um jogador de Go desenvolvido com os recursos oferecidos pelo Smart Game Board. Em 2007, motivado pelo sucesso alcançado pelos programas jogadores de Go, CrazyStone e MOGO, que utilizam o método de busca em árvore Monte-Carlo, Enzenberger começou a desenvolver um programa que implementasse tal técnica [Enzenberger et al. 2010]. Este programa foi desenvolvido utilizando recursos do Smart Game Board e as principais rotinas que tratavam de Go do jogador Explorer. Inicialmente chamado apenas de UCT, o jogador foi renomeado para Fuego e se tornou um projeto código-aberto em 2008. A principal inovação do Fuego é o fato de que pela primeira vez uma implementação das técnicas mais atuais do estado da arte está disponível de maneira gratuita e acessível para qualquer pessoa estudar e modificar. Isto é feito na forma de um software código-aberto consistente, bem projetado e testado. A Fuego é uma plataforma de software e não apenas uma biblioteca com funcionalidade limitada. Um grande número de classes e funções são encontrados na plataforma que está dividida em cinco bibliotecas. A Figura 3.1 a seguir apresenta o grafo de dependência entre os principais componentes.

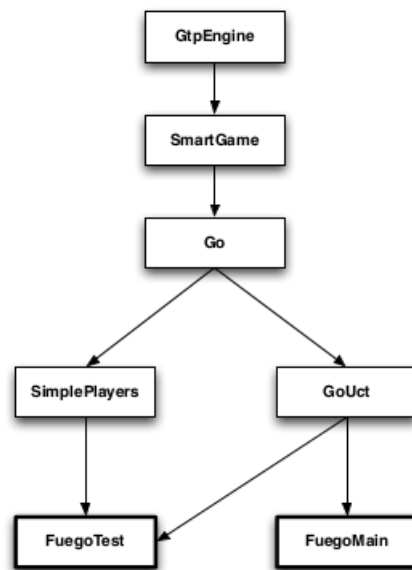


Figura 3.1: Principais componentes da plataforma Fuego e suas dependências.

Cada componente da plataforma Fuego é descrito a seguir:

- a biblioteca *GtpEngine* oferece uma implementação abstrata do protocolo GTP (*Go Text Protocol*). Este protocolo foi usado pela primeira vez pelo programa GnuGo [FSF 2013] e ganhou ampla aceitação na comunidade desenvolvedora de programas para Go. O protocolo foi posteriormente adaptado para outros jogos. Basicamente, a biblioteca *GtpEngine* recebe e interpreta os comandos externos, invocando as rotinas necessárias associadas a cada comando;
- a biblioteca *SmartGame* contém funcionalidades úteis a jogos em geral, como classes e funções que ajudam a representar o estado de tabuleiro de um jogo de dois jogadores. As classes mais complexas contidas nesta biblioteca são as implementações da busca *alpha-beta* e a busca em árvore Monte-Carlo. Ambas rotinas são implementadas independentemente do domínio do jogo;
- a biblioteca *Go* é implementada a partir da *SmartGame* e oferece funcionalidades específicas do jogo Go. A classe mais importante é uma implementação eficiente do tabuleiro de Go, comentada em maiores detalhes na Subseção 3.1.2 a seguir;
- o módulo *SimplePlayers* reúne algoritmos simples de jogadores automáticos de Go como, por exemplo, um jogador que faz movimentos aleatórios e outro que procura maximizar uma função de avaliação. Estes jogadores podem ser utilizados para testes e como referência para a implementação de novos jogadores;
- os módulos *GoUCT* e *FueGoMain* são responsáveis por implementar a parte principal do motor do Fuego e;
- finalmente, o módulo *FueGoTest* é uma interface para funcionalidades presentes nas bibliotecas *SmartGame* e *Go*, sendo utilizada principalmente para testes.

Vários programas jogadores de Go e outros jogos utilizam a plataforma Fuego. Entre estes programas destacam-se o MoHex [Arneson et al. 2009], um programa jogador do jogo Hex de nível profissional, o TsumeGo Explorer [Kishimoto 2005] que é o melhor solucionador do problema de vida e morte do jogo de Go e o jogador RLGO [Silver et al. 2007] que aprende formas locais por diferenças temporais com o mínimo de conhecimento específico de Go. O jogador Fuego utiliza a busca em árvore Monte-Carlo e o algoritmo UCT, explicados na Seção 2.3. A implementação da busca do Fuego possui várias otimizações como o reuso da subárvore de uma busca anterior. Neste processo, ao iniciar a busca por um movimento, é verificado se é possível reutilizar os nós da subárvore construída pela busca do movimento anterior. Isto permite que a Fuego reutilize os valores calculados para os nós da subárvore anterior. O processo de busca empregado pelo agente é guiado por heurísticas do jogo de Go codificadas por humanos. Além disso, para avaliar novos nós encontrados durante a construção da árvore de busca, o Fuego também utiliza heurísticas, conforme explicado em maiores detalhes na Subseção 3.1.1. O jogador conta também com um livro de aberturas que contém jogadas iniciais de boa qualidade. Estes aspectos caracterizam a abordagem fortemente supervisionada adotada pelo Fuego.

3.1.1 *Prior Knowledge*

Durante a busca MCTS executada pelo agente Fuego, as heurísticas, além de serem utilizadas para guiar as simulações Monte-Carlo, são também empregadas para obter uma avaliação prévia de novos nós encontrados durante a fase de construção da árvore. No momento da expansão, para cada nó s_i , gerado através da execução da ação a a partir de s , o agente Fuego emprega heurísticas codificadas por especialistas humanos para obter o valor do termo de qualidade $Q^t(s, a)$ da equação 2.4. No contexto do jogador Fuego, esta estratégia é denominada *Prior Knowledge* e tem como objetivo avaliar previamente os nós gerados na expansão, sem executar uma simulação a partir destes. O valor obtido pelo *Prior Knowledge* é usado pelo algoritmo UCT para escolher um dentre os novos nós gerados. Desta maneira, é possível utilizar as simulações de forma mais eficiente, explorando nós cujos valores iniciais são mais promissores. As heurísticas utilizados pelo agente Fuego consistem na verificação de várias situações de tabuleiro como por exemplo o *atari*, explicado anteriormente na Seção 2.1. Um outro exemplo de uma situação em que o uso das heurísticas do *Prior Knowledge* é bastante relevante é apresentado a seguir. Conforme discutido anteriormente, sempre que um nó possui filhos ainda não visitados, estes têm a preferência sobre outros que já foram visitados pelo menos uma vez. Assim, um episódio de simulação sempre é executado a partir de um nó desconhecido, mesmo que este corresponda a uma ação obviamente muito ruim como colocar em *atari* (iminência de captura) um grupo de peças que pertence ao próprio jogador, conforme é mostrado no exemplo da Figura 3.2 a seguir.

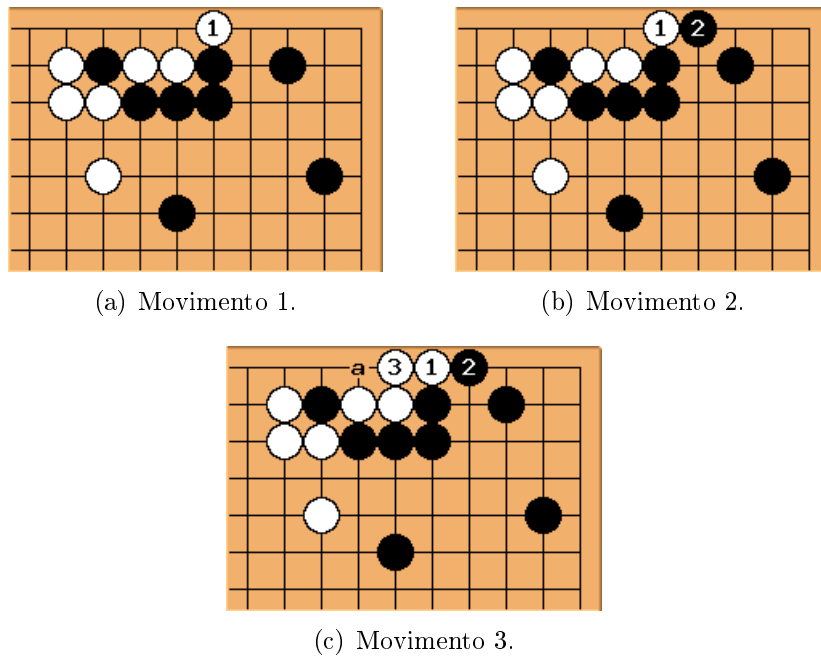


Figura 3.2: Exemplo de *auto atari*: devido a peça branca **3**, todo o grupo pode ser capturado com uma peça preta na posição **a**.

O grupo de peças brancas cercado pelas peças pretas tem poucas liberdades e corre sério risco de ser capturado. O movimento **1** das peças brancas reduz uma liberdade do grupo de peças pretas e ameaça uma *fuga*, conectando o grupo de peças brancas com o exterior. Após a resposta das peças pretas com o movimento **2**, a peça branca **3** representa uma péssima jogada pois imediatamente coloca todo o seu grupo em *atari*. Em outras palavras, após o posicionamento da peça **3**, todo o grupo de peças brancas é facilmente capturado se uma peça preta é colocada no ponto **a** indicado no tabuleiro. A este tipo de movimento, exemplificado por **3**, denomina-se *auto-atari*, pois o próprio jogador coloca suas peças em *atari*, facilitando a captura pelo oponente. Assim, mesmo que a ação correspondente a um nó seja claramente ruim, como o *auto-atari* mostrado anteriormente na Figura 3.2, supondo que o *Prior Knowledge* não seja utilizado, na primeira vez que o nó é visitado e adicionado à árvore, um episódio de simulação será executado para se obter uma primeira avaliação. Entretanto, ao empregar a abordagem de *Prior Knowledge*, os nós visitados pela primeira vez durante a construção da árvore são avaliados previamente, sem executar uma simulação a partir destes. A estratégia *Prior Knowledge* verifica a presença de vários padrões de peças, além do *atari* e o *auto-atari*, mostrados anteriormente. A partir destas heurísticas, é possível atribuir um valor inicial aos nós até então desconhecidos, o que permite evitar a escolha aleatória pelo algoritmo UCT. Desta maneira, é possível utilizar as simulações de forma mais eficiente, explorando nós cujos valores iniciais são mais promissores. Conforme será apresentado posteriormente, uma das distinções entre os agentes SDM-Go e Fuego, no que diz respeito à implementação da busca MCTS, reside no processo de avaliação prévia (*Prior Knowledge*) dos nós desconhecidos gerados na fase de

construção da árvore. Enquanto no agente Fuego esta estratégia é fortemente dependente do domínio, por empregar heurísticas específicas do jogo de Go, o SDM-Go, por outro lado, concentra seu esforço em substituir esta estratégia por outra não supervisionada, utilizando a memória SDM como recurso adicional, conforme explicado no capítulo 4.

3.1.2 Representação de Tabuleiro do Agente Fuego

A classe que implementa o tabuleiro de Go do Fuego é uma das mais importantes na arquitetura do agente. A implementação do tabuleiro é bastante útil, mantendo um histórico dos movimentos executados, o que permite desfazê-los posteriormente [Enzenberger et al. 2010]. Esta funcionalidade é utilizada frequentemente, pois, na árvore construída durante a busca, os nós correspondem apenas a ações, ou movimentos, no tabuleiro. Assim, para obter o estado terminal do tabuleiro após finda a fase de *playout*, é necessário executar as ações correspondentes aos nós deste caminho. Isto ocorre antes da fase de retropropagação, quando a sequência de ações que corresponde ao caminho percorrido na árvore e as ações da fase de *playout*, após executadas, geram o tabuleiro final atingido pelo episódio. O resultado do *playout* é obtido e retropropagado para atualização dos valores dos nós da árvore. Em seguida, as ações são desfeitas, retornando o tabuleiro ao estado original em que se encontrava no início da busca. Além disto, a implementação do agente Fuego também mantém a informação sobre as peças e as liberdades de cada grupo no tabuleiro e verifica a legalidade dos movimentos. Diferentes convenções de final de jogo, como remover ou não as peças capturadas, também estão implementadas. Todas estas funcionalidades oferecidas pela representação de tabuleiro do agente Fuego são eficientes e contribuem para o bom funcionamento deste jogador, não interferindo no tempo de processamento da busca MCTS. Entretanto, o uso deste tabuleiro pelo agente SDM-Go se torna inviável, no que se refere ao armazenamento e tratamento de vários estados diferentes na memória SDM. Além disto, para a operação da memória esparsamente distribuída quase todas estas funcionalidades do tabuleiro utilizado pelo agente Fuego são desnecessárias. De fato, a memória do SDM-Go necessita apenas saber a posição e a cor de cada peça e finalmente ser capaz de comparar estas duas informações entre os vários tabuleiros armazenados. A representação de tabuleiro requerida é, portanto, bastante simples, conforme apresentado na Seção 4.1.2.

3.2 RLGO

O RLGO [Silver et al. 2007] é um jogador de Go de código-aberto baseado nos recursos oferecidos pela plataforma do jogador Fuego. O RLGO utiliza um processo de aprendizado não supervisionado, baseado em redes neurais. Empregando o método das diferenças temporais (TD), o RLGO aprende formas locais, ou padrões de posicionamento

de peças, no tabuleiro. Além disso, o programa também incorpora os algoritmos TD-search e Dyna-2 [Silver e Müller 2012] que combinam busca e aprendizado. No RLGO, o cálculo da função que avalia os tabuleiros (função de avaliação) é simplificado pela detecção de semelhanças entre tabuleiros, em virtude de repetição de posicionamento de peças (padrões). Em outras palavras, tabuleiros contendo os mesmos padrões de outros, e portanto já avaliados, terão seus cálculos simplificados. A semelhança ou equivalência entre os tabuleiros é estabelecida por heurísticas de classes de equivalência que levam em consideração a rotação, espelhamento, translação ou inversão de cores dos padrões. Durante o jogo, um estado de tabuleiro de tamanho $N \times N$ pode ser representado por um vetor de variáveis, uma para cada um dos N^2 pontos, que podem assumir um de três valores distintos: branco, preto ou vazio. Então, uma forma local corresponde a uma configuração específica destas variáveis em uma dada região $K \times K$ do tabuleiro, conforme ilustrado na Figura 3.3. O RLGO utiliza todas as configurações de formas locais de tamanho 1×1 , 2×2 e 3×3 para construir um longo vetor de características. Cada região quadrada do tabuleiro de 1×1 até 3×3 é visitada e varrida para identificar quais formas locais ali se encontram. Se uma forma local L_i está presente no tabuleiro, sua posição correspondente no vetor de características é ativada através do valor 1 e em caso contrário, se L_i não se encontra no tabuleiro, o valor é 0. O vetor é bastante esparsa, pois exatamente uma forma local é encontrada em cada região quadrada $K \times K$, com $1 \leq K \leq 3$. De maneira a aproveitar as simetrias do tabuleiro de Go (rotação, espelhamento, translação e inversão de cores), as formas locais são utilizadas com compartilhamento de pesos [Schraudolph et al. 1994]. As relações de equivalência entre as formas locais, devido a rotação e reflexão por exemplo, definem classes de formas que podem ser representadas unicamente pela forma local com menor índice j , chamada de exemplo canônico da classe. Assim, cada forma local i em uma classe de equivalência compartilha os mesmos valores θ_j do exemplo canônico j . Além disso, dois vetores de valores θ são utilizados:

- um dos valores, θ^{LD} , depende da posição absoluta da forma local no tabuleiro. Desta maneira, as formas locais numa mesma classe de equivalência compartilham o mesmo valor θ^{LD} que suas equivalentes rotacionados, refletidos ou de cor invertida;
- o outro valor, θ^{LI} , não considera a posição absoluta da forma local. Neste caso, em uma mesma classe, as formas locais compartilham o mesmo valor θ^{LI} com suas equivalentes simétricas em diferentes posições do tabuleiro.

A Figura 3.3 exemplifica essas duas formas de compartilhamento de valores. Assim, o RLGO utiliza o método das diferenças temporais para aprender uma função de avaliação baseada nos valores das formas locais ativadas no vetor de características. Em 2012, o RLGO foi melhorado com a implementação do algoritmo *TD-search* [Silver e Müller 2012] que combina busca e aprendizado. A principal ideia deste método é especializar uma função de avaliação global transformando-a em uma poderosa função local. A partir de

cada configuração corrente de tabuleiro, a função global é treinada utilizando experiência obtida de várias simulações. Esta função é atualizada em tempo real e se torna cada vez mais especializada nas táticas e estratégias relevantes para a situação do jogo corrente. Assim como na busca MCTS, o algoritmo *TD-search* utiliza as simulações para melhorar as estimativas dos valores dos estados de tabuleiro. Entretanto, de maneira diferente, a árvore de busca que armazena individualmente os valores dos estados de tabuleiro é substituída por uma função de avaliação baseada em características aprendida pelo método TD.

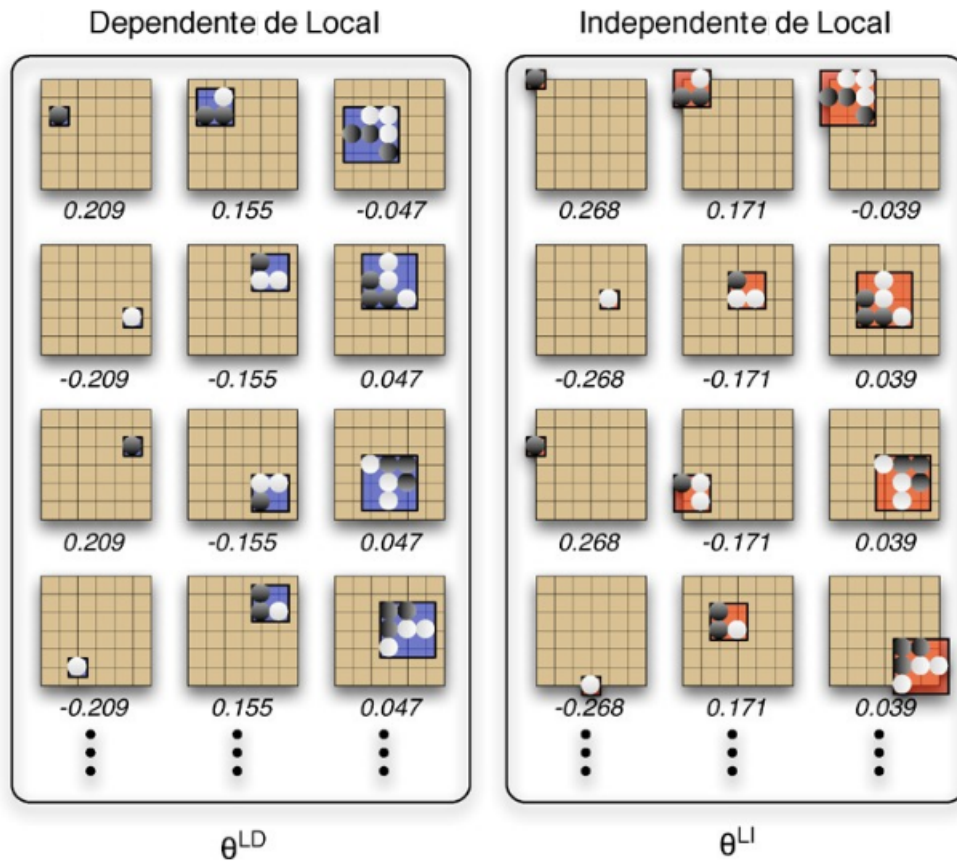


Figura 3.3: Exemplos de formas locais utilizadas pelo RLGO.

O SDM-Go, assim como o RLGO, também utiliza os resultados das simulações da busca MCTS através de um método de aprendizado não supervisionado. Conforme detalhado no capítulo 4, os resultados das simulações Monte-Carlo são aproveitados pelo SDM-Go para ajustar os valores utilizados em uma função avaliação dos nós da árvore de busca.

3.3 Honte

O jogador de Go Honte [Dahl 1999] utiliza redes neurais juntamente com outras técnicas convencionais de inteligência artificial como a busca *alpha-beta*. A ideia buscada

neste jogador é balancear o uso de aprendizado de máquina com outras técnicas de programação. Desta maneira, espera-se que o jogador automático se beneficie com o melhor dos dois mundos. A partir de uma visão geral, o processo de avaliação de jogadas do Honte segue os seguintes passos:

1. calcula quais grupos podem ser capturados (busca *alpha-beta*);
2. calcula quais grupos podem ser conectados (busca *alpha-beta*);
3. identificação do conjunto de grupos, de acordo com suas conexões (grafo);
4. calcula vida ou morte dos grupos (busca *alpha-beta*);
5. cálculo de influência (grafo);
6. avaliação da segurança de um grupo (rede neural);
7. avaliação de território (rede neural);
8. somatório das estimativas de território e segurança dos grupos.

A execução destes passos demanda bastante tempo de processamento e uma avaliação completa não pode ser executada na maioria das jogadas. De maneira a reduzir o número de jogadas a serem avaliadas, uma rede neural é treinada para identificar boas configurações de peças. Um livro de jogadas, contendo padrões chamados *joseki*, também é utilizado para fornecer possíveis movimentos. A rede neural que identifica bons padrões de posicionamento de peças é treinada com base em um conjunto de 400 jogos de especialistas. O objetivo da rede é identificar se uma determinada jogada seria feita por um jogador humano especialista. A entrada que alimenta a rede é uma configuração local de peças próxima a uma intersecção vazia do tabuleiro. A saída é interpretada como a qualidade de se posicionar uma peça neste ponto vazio. O treinamento da rede consiste em selecionar aleatoriamente, do conjunto de 400 jogos, uma posição do tabuleiro. A jogada feita pelo especialista produz o resultado esperado, com valor de saída igual a 1. Uma jogada aleatória em outro ponto do tabuleiro produz a jogada com valor de saída igual a 0. Para avaliar todo o tabuleiro, a busca *alpha-beta* não é uma alternativa prática. Porém para problemas locais, bem definidos, ela se torna uma importante ferramenta. No Honte, ela é utilizada para decidir sobre a possibilidade de capturar um grupo de peças, a possibilidade de conexão entre dois grupos de peças e o problema de vida e morte de um grupo. O resultado da busca é também aplicado para determinar se é possível impedir o oponente de realizar uma destas três possibilidades. No Honte, uma busca *alpha-beta* genérica é empregada, retornando um de três resultados: sim, não e desconhecido. A busca utiliza aprofundamento iterativo e uma tabela *hash* como tabela de transposição para armazenar estados já visitados. Um conceito importante no jogo Go é a influência de um grupo de peças. Uma boa influência dificulta a formação de grupos pelo inimigo e facilita o estabelecimento de território do jogador. O cálculo de influência feito pelo

agente Honte é inspirado em processos estocásticos e pode ser visto como uma caminhada aleatória: para um ponto p do tabuleiro, inicie uma caminhada aleatória, com mesma probabilidade de mover-se para qualquer uma das adjacências. A caminhada termina quando uma peça é encontrada ou após vinte passos. Define-se então a influência do jogador das peças de cor preto como a probabilidade de terminar numa peça desta cor. Similarmente, a influência das peças de cor branco é calculada da mesma forma. Quando um jogador humano avalia uma posição de tabuleiro, primeiramente decide quais peças fazem parte do mesmo grupo e então avalia a segurança desses grupos. Na maior parte do tempo, não é claro decidir se um grupo irá sobreviver ou não e os jogadores humanos confiam na experiência e intuição para decidir sobre a segurança dos grupos [Lee et al. 2009]. O jogador Honte utiliza uma rede neural para avaliar as chances de sobrevivência de um grupo. A rede recebe parâmetros do grupo como tamanho, número de liberdades e influência como entrada. A saída produzida é uma estimativa da probabilidade do grupo de se manter até o final do jogo. A rede aprende através das diferenças temporais, jogando contra um clone dela mesma. A rede utiliza treze neurônios de entrada e 20 neurônios na camada escondida. A saída é um número real entre 0 e 1. Ainda uma terceira rede neural é treinada para avaliar a probabilidade de um ponto vazio pertencer, no final do jogo, ao jogador das peças de cor preto ou de cor branco. Esta rede utiliza como entrada apenas os valores de influência calculados para ambos jogadores e a distância até as bordas do tabuleiro. O treinamento desta rede é também através das diferenças temporais e por *self-play*. O agente Honte é um bom exemplo de um jogador de Go com níveis satisfatórios de desempenho que utiliza recursos de aprendizado combinados com técnicas tradicionais de busca, como a *alpha-beta*. Assim como o Honte, o SDM-Go combina técnicas de aprendizado e algoritmos de busca, como apresentado no capítulo 4.

3.3.1 NeuroGo

O jogador NeuroGo [Enzenberger 2003] implementa uma rede neural que procura integrar conhecimento especialista do jogo Go. A rede neural aprende jogando contra si mesma (*self-play*) e utiliza o aprendizado por diferenças temporais e retropropagação. Esta abordagem permitiu que o NeuroGo atingisse níveis de jogo comparáveis com os programas convencionais, que utilizam fortemente bases de dados e conhecimento especialista codificado manualmente. De maneira geral, o jogador recebe uma configuração de tabuleiro e responde com uma estimativa sobre cada intersecção do tabuleiro. Esta estimativa é a probabilidade de que tais pontos no tabuleiro pertencerão ao jogador no final do jogo. O sistema contém uma rede neural com apenas uma camada escondida e funções de ativação sigmóides. O conhecimento especialista de Go é inserido na rede através de três sistemas especialistas, descritos a seguir. Primeiramente, antes de alimentar a rede com a configuração do tabuleiro, esta é transformada. O sistema sempre considera que o jogador

das peças de cor preto está jogando. Ou seja, caso seja o a vez do jogador com as peças de cor branco, as cores são invertidas. Posteriormente, o programa não considera os grupos de peças conectadas e as avalia como uma unidade individual. Isto decorre do fato de que se um grupo de peças sobrevive até o final do jogo, então todas as peças sobrevivem. Caso contrário, todas as peças do grupo são capturadas e o espaço que elas ocupam se torna território inimigo. Assim, a configuração do tabuleiro é transformada em um conjunto de grupos e intersecções vazias. A transformação inicial do tabuleiro determina o número de neurônios da rede. As conexões entre estes não são portanto estáticas e variam de acordo com a configuração do tabuleiro. Um sistema especialista detecta relações entre os grupos ou unidades, gerados pela transformação do tabuleiro. Um grafo é construído usando as unidades como nós e as relações como arestas. A conectividade deste grafo define as ligações dos neurônios da rede. Dois neurônios de camadas adjacentes estão conectados somente se as unidades que representam estão relacionadas. Os pesos utilizados nas conexões dependem do tipo de relação, tipo da unidade e da posição dos neurônios nas camadas. Isto incorpora uma independência de localização das unidades, traduzindo-se em invariância rotacional e translacional. O segundo especialista do NeuroGo identifica características das unidades e alimenta a rede com essas informações, ao invés de apenas fornecer a posição e relacionamento das unidades. As características identificadas pelo sistema especialista são informações pertinentes a táticas e estratégias do jogo. Algumas são o número de liberdades de um grupo e o número de peças do grupo, por exemplo. E ainda um outro sistema especialista avalia partes do tabuleiro por si próprio, sem intervenção ou comunicação com a rede. Este sistema pode ser usado em casos onde existe um algoritmo simples e correto que avalia e decide sobre certa classe local de posições de peças. O resultado deste especialista, caso ativado, sobrepõe-se a avaliação da rede neural no momento da escolha do movimento. Entretanto, ele é utilizado, de maneira suavizada, para o aprendizado da rede neural. O SDM-Go, assim como o NeuroGo também utiliza técnicas de aprendizado sobre uma representação do tabuleiro própria, como descrito no capítulo 4.

Capítulo 4

SDM-Go

Este capítulo apresenta o SDM-Go (*Sparse Distributed Memory - Go*), um agente para o jogo de Go que tem como objetivo introduzir técnicas de aprendizado no jogador Fuego utilizando o modelo de memória esparsamente distribuída. Para alcançar esta meta, o SDM-Go estende a arquitetura do Fuego com a inclusão de dois módulos:

1. SDM: o módulo principal da memória esparsamente distribuída, onde são armazenados os estados de tabuleiros já visitados juntamente com suas respectivas avaliações calculadas pelo sistema. Os estados armazenados na SDM serão referenciados no escopo deste trabalho como **protótipos**;
2. BBR (*Bit Board Representation*): o módulo responsável pela transformação da representação do tabuleiro utilizada pelo agente Fuego para a forma de vetores de bits, utilizada pelo agente SDM-Go. Esta representação, além de proporcionar um armazenamento eficiente dos tabuleiros, reduzindo o espaço físico de memória necessário, também possibilita a rápida execução das operações envolvendo novos estados de tabuleiros visitados e os protótipos já memorizados.

Conforme será apresentado nesta Seção, a proposta do SDM-Go consiste em utilizar a memória SDM para reduzir o caráter supervisionado da busca MCTS executada pelo jogador Fuego, no que diz respeito ao uso da estratégia de *Prior Knowledge* (*Conhecimento Prévio*) como valor de base do termo $Q^t(s, a)$ da equação 2.4 para os nós que se candidatam à inserção na árvore de busca. Esta estratégia utiliza heurísticas codificadas por especialistas humanos para avaliar as posições do tabuleiro correspondentes a cada movimento. Por sua vez, o SDM-Go se diferencia exatamente na maneira de calcular o valor base do termo $Q^t(s, a)$, uma vez que ele substitui tais heurísticas por valores obtidos no histórico de tabuleiros similares previamente avaliados (que estão armazenados na memória SDM), inclusive daqueles relativos aos movimentos anteriores ocorridos no mesmo episódio de busca. Dessa forma, ao utilizar tal histórico, o SDM-Go, distintamente do Fuego, não desperdiça o valor do conhecimento expresso nessas informações. Nessa

direção, na parte dos testes avaliativos, diversos cenários serão investigados, ora usando, ora retirando o *prior knowledge*.

4.1 Arquitetura Geral

O agente SDM-Go é baseado no agente Fuego e melhora o processo de busca deste através da utilização da memória SDM como técnica de aprendizado. O SDM-Go é formado de três módulos principais, conforme apresentado na arquitetura geral da Figura 4.1. Assim como o jogador Fuego, o SDM-Go constrói uma árvore de busca Monte-Carlo utilizando o algoritmo UCT. Entretanto, diferentemente do Fuego, o módulo de busca do SDM-Go se comunica com outros dois que compõe a arquitetura básica do agente. Estes dois outros módulos da arquitetura do SDM-Go são o módulo de representação em vetores de bits (*Bit Board Representation - BBR*) e o módulo da memória esparsamente distribuída (SDM). A Figura 4.1 apresenta os módulos que compõe a arquitetura do SDM-Go e os relacionamentos entre eles. Durante a fase de construção da árvore da busca pelo melhor movimento, novos nós s_i filhos de um nó s , que não pertencem à árvore são encontrados. No caso do agente Fuego uma avaliação prévia destes nós é obtida através de heurísticas do jogo de Go (*Prior Knowledge*). De maneira distinta, o agente SDM-Go utiliza informações de jogos anteriores, através dos estados de tabuleiro armazenados na memória SDM. Assim, no caso do SDM-Go o módulo de busca MCTS comunica-se com a SDM, primeiramente passando o nó s e os movimentos geradores dos filhos s_i para o módulo de representação do tabuleiro em vetores de bits (BBR), conforme indicado pela seta #1a na Figura 4.1. O módulo BBR converte a representação de tabuleiro utilizada pelo módulo de busca MCTS para outra representação baseada em vetores de bits. Com base nesta nova representação do nó s , a representação em bits dos nós filhos s_i é rapidamente construída utilizando operações *bitwise* (ver Seção 2.5). A partir daí, as representações dos nós são passadas para o módulo da SDM, o que é indicado pela seta #1b. Novamente utilizando operações *bitwise*, cada nó s_i passa por um teste de similaridade com os nós previamente armazenados na memória SDM. Caso o teste seja bem sucedido, s_i é avaliado de acordo com os valores dos nós similares e o resultado é informado ao módulo de busca MCTS, como mostra a seta #2 da Figura 4.1. Contando agora com a avaliação prévia fornecida pela SDM, a busca Monte-Carlo prossegue normalmente. Ao final do episódio de simulação, durante a fase de retropropagação, o resultado do final do jogo é usado para atualizar os valores dos nós pertencentes ao caminho percorrido na árvore. Neste momento, além de executar esta atualização, o SDM-Go também aproveita para atualizar os valores dos nós armazenados na memória SDM. Para tanto, o módulo de busca MCTS comunica o resultado e o conjunto S dos nós do caminho percorrido do episódio de simulação corrente para o módulo de memória SDM, como indicado pela seta #3 na Figura 4.1. Mais uma vez, esta comunicação é intermediada pelo módulo BBR

que converte a representação de tabuleiro usada no módulo MCTS para a representação em bits, adequada ao processamento da memória SDM. Através de operações bitwise, são selecionados aqueles protótipos armazenados na memória que atendem ao critério mínimo de similaridade com os nós do conjunto S , pertencentes ao caminho percorrido na árvore. Estes protótipos são então atualizados de acordo com o grau de similaridade com os nós de S . As próximas seções descrevem em detalhes o funcionamento de cada módulo da arquitetura para maior entendimento deste processo que foi apresentado de forma geral.

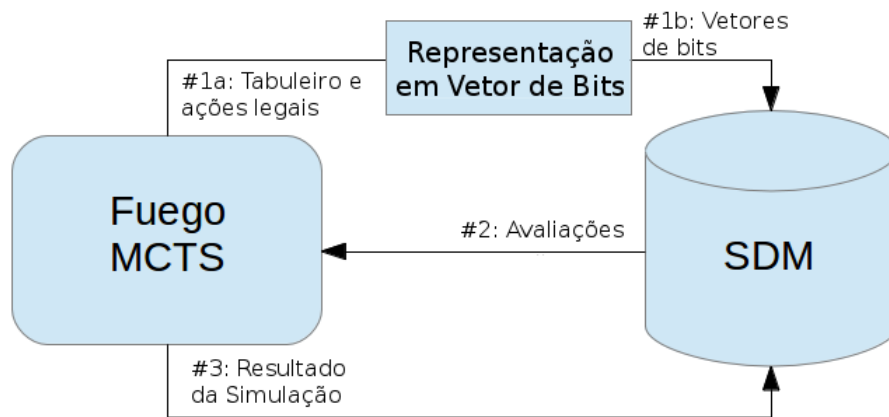


Figura 4.1: Arquitetura Geral do SDM-Go

4.1.1 Módulo de Busca em Árvore Monte-Carlo

Como no Fuego, a busca MCTS implementada no agente SDM-Go considera as ações como os nós da árvore, ao contrário de outras abordagens que consideram a configuração do tabuleiro. No jogo de Go, uma ação corresponde ao posicionamento de uma peça no tabuleiro. Assim, a árvore de busca construída pela busca do SDM-Go tem sua raiz no estado corrente do tabuleiro. Os filhos do nó raiz da árvore são as ações possíveis a partir do estado corrente de tabuleiro. Os nós do nível seguinte de profundidade são correspondentes às ações possíveis a partir dos estados de tabuleiro atingidos através de cada ação do nível anterior e assim, sucessivamente, até os nós folhas que representam a ação **nulo**. O movimento nulo significa que o jogo atingiu um estado final e que o tabuleiro resultante da sequência de ações do episódio pode ser avaliado como indicador do resultado do jogo (vitória, derrota ou empate). A ação **nulo** é alcançada após duas ações consecutivas de **passar a vez** ou, ainda, quando não é mais possível posicionar alguma peça em algum ponto do tabuleiro. Em jogos de profissionais, em um certo momento da partida, ambos jogadores reconhecem qual será o resultado do jogo e, através da ação de **passar**, indicam que não é necessário mais executar movimento algum. A Figura 4.2

apresenta um exemplo de uma árvore de busca construída pela implementação do agente SDM-Go a partir do estado de tabuleiro mostrado.

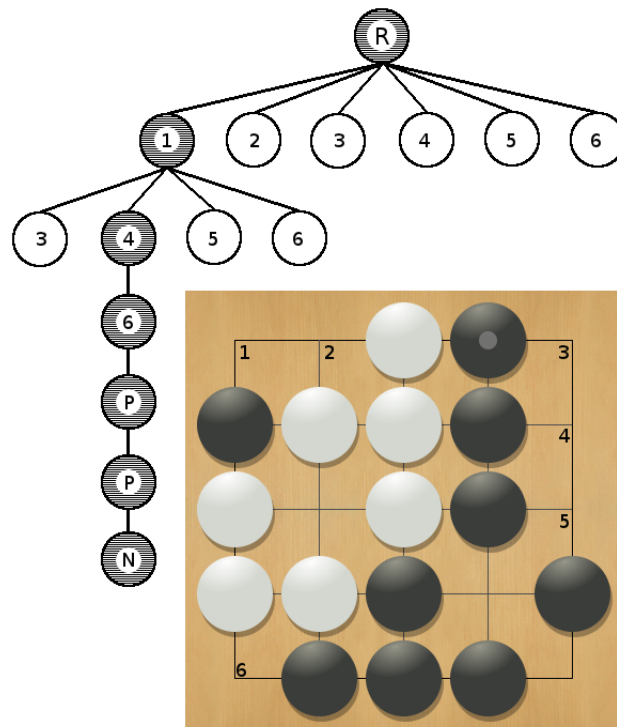


Figura 4.2: Árvore de busca para configuração de tabuleiro 5x5.

A raiz da árvore **R** corresponde à configuração de tabuleiro apresentada na mesma Figura, sendo a vez do jogador com as peças brancas realizar um movimento. Os números de 1 a 6 marcam as posições do tabuleiro onde é possível posicionar uma peça branca. Assim, os nós filhos do nó raiz correspondem, respectivamente, às ações de posicionar uma peça em cada uma das posições de 1 a 6. Suponha que a partir da expansão inicial do nó raiz, o nó candidato **1** seja escolhido para expansão, por ser o filho de melhor avaliação. A partir deste nó, o jogador das peças pretas pode posicionar peças nas posições de 3 a 6. Neste nível de profundidade, sendo vencedor para ser expandido o nó **4**, seu filho único, o nó **6**, corresponde à única possibilidade de ação para as peças brancas. A partir deste nó, as únicas ações possíveis para os dois lados são representadas pelo nó **P**, que significa **passar a vez**, imediatamente seguida pelo movimento nulo, correspondido pelo nó **N**, indicando o final do jogo. É importante ressaltar que, no exemplo da Figura, em qualquer momento do jogo é possível executar a ação de passar a vez. Além disso, existem outros dois pontos que poderiam receber peças pretas. Entretanto, estes foram omitidos da árvore Figura 4.2, de maneira a simplificar o exemplo, bem como apresentar uma situação mais verossímil. Por fim, para obter a configuração de peças do estado de final de jogo é necessário executar as ações do caminho percorrido na árvore, já que esta não armazena os estados de tabuleiro, mas apenas as ações possíveis. Ao executar as ações associadas aos nós visitados no caminho indicado na Figura 4.2, obtém-se o estado

final de jogo mostrado a seguir na Figura 4.3. O resultado é vitória para o jogador com as peças brancas por 5.5 pontos (considerando um *komi* de 6.5). O resultado de vitória para as peças brancas será então retropropagado pelo caminho na árvore, atualizando os valores dos nós envolvidos.

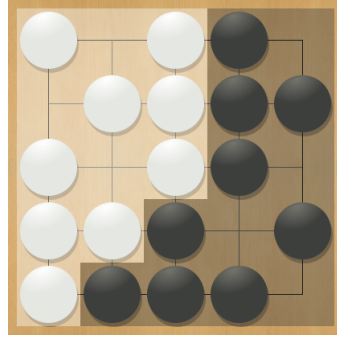


Figura 4.3: ConFiguração final de jogo no tabuleiro 5x5.

A busca implementada pelo SDM-Go é, então, centrada na construção de uma árvore que segue o modelo apresentado, no qual os nós correspondem a ações ou movimentos do jogo em oposição a estados do tabuleiro. O pseudocódigo mostrado na Figura 4.4 a seguir detalha a implementação da busca no agente SDM-Go.

BuscaMCTS()

```

1: IniciaThreadsArvore()
2: ContinuaBusca = VERDADEIRO
3: while ContinuaBusca do
4:   EstadoFinal = FALSO
5:   SeqAcoes = PercorreArvore(EstadoFinal)
6:   TamanhoCaminho = Tamanho(SeqAcoes)
7:   if EstadoFinal = VERDADEIRO then
8:     SimulaJogo(SeqAcoes)
9:   end if
10:  TamanhoSimulacao = Tamanho(SeqAcoes)
11:  TabuleiroFinalJogo = Executa(SeqAcoes)
12:  Resultado = Avalia(TabuleiroFinalJogo)
13:  DesfazAcoes(TamanhoSimulacao – TamanhoCaminho, SeqAcoes)
14:  DesfazAcoes(TamanhoCaminho, SeqAcoes)
15:  AtualizaArvore(Resultado, SeqAcoes)
16:  if TerminaBusca() then
17:    ContinuaBusca = FALSO
18:  end if
19: end while
20: return MelhorAcao()

```

Figura 4.4: Pseudocódigo da busca em árvore Monte-Carlo implementada pelo agente SDM-Go.

As instruções do pseudocódigo apresentado na Figura 4.4 são comentadas a seguir.

Linha 1. A árvore de busca é iniciada, colocando o estado de tabuleiro corrente no

nó raiz. Caso seja possível aproveitar nós que foram previamente explorados em outras execuções da busca durante a partida, estes também são adequadamente inseridos na árvore. Além disso, caso estejam habilitadas, múltiplas *threads* são iniciadas para acelerar a busca.

Linha 5. O procedimento *PercorreArvore* é invocado. Este procedimento é responsável pela fase de **construção da árvore** (**seleção e expansão**) da busca MCTS. A variável *booleana EstadoFinal* é passada como argumento e dentro do procedimento receberá um valor indicando se algum estado final foi atingido devido ao caminho percorrido na árvore. Ao final, o procedimento retorna a sequência de nós (ações) que foram selecionados, juntamente com o nó recém adicionado na árvore.

Linha 6. O tamanho do caminho percorrido na árvore é armazenado para ser usado posteriormente durante a busca.

Linha 7. Verifica-se se algum estado final foi atingido durante a fase de construção da árvore, executadas anteriormente no procedimento *PercorreArvore*.

Linha 8. O procedimento *SimulaJogo*, que irá executar a fase de **playout** da busca, recebe a sequência de ações como argumento. A partir do último nó desta sequência, que foi adicionado anteriormente na fase de construção da árvore, o jogo será conduzido até um estado final.

Linha 10. O tamanho da sequência de ações após a execução do procedimento *SimulaJogo* é armazenado. Caso o procedimento não tenha sido executado, devido ao teste da linha 7, este tamanho tem o mesmo valor do armazenado anteriormente na variável *TamanhoCaminho*.

Linha 11. O estado do tabuleiro no final de jogo, alcançado anteriormente através dos procedimentos *PercorreArvore* ou *SimulaJogo*, é obtido através da execução da sequência de ações a partir do estado corrente do tabuleiro.

Linha 12. O resultado do tabuleiro final do jogo é calculado e armazenado na variável *Resultado*.

Linha 13. As ações da fase de *playout* são desfeitas de trás para frente. O número de ações a serem desfeitas é indicado através da subtração entre os valores de tamanho obtidos nos procedimentos *PercorreArvore* e *SimulaJogo*, armazenados nas variáveis *TamanhoCaminho* e *TamanhoSimulacao* respectivamente. Caso, nenhuma ação tenha sido acrescentada durante o *playout*, os dois valores de tamanho são iguais e o resultado da subtração é zero. Assim, obtém-se o número exato de ações a serem desfeitas devido à fase de *playout*.

Linha 14. Da maneira similar à instrução anterior, as ações correspondentes aos nós do caminho percorrido na árvore também são desfeitas, de trás para frente, restaurando o tabuleiro ao estado a partir do qual a busca foi iniciada.

Linha 15. O resultado do tabuleiro do final do jogo é retropropagado, seguindo a sequência de ações selecionadas no caminho percorrido na árvore, para atualizar os valores

destes nós.

Linha 16. Verifica-se se é possível prosseguir com a busca. A implementação da busca permite que vários parâmetros sejam configurados para sinalizar seu fim, como limite de **tempo** e limite de número de **episódios**.

Linha 20. Ao final do procedimento, a ação correspondente ao *melhor* nó filho da raiz é retornada. Neste caso, a implementação permite diferentes critérios para decidir qual é o *melhor* nó:

1. Maior valor (taxa de vitórias durante os episódios da busca);
2. Maior contagem (número de vezes em que o nó foi visitado durante os episódios da busca);
3. Maior valor UCT (valor da fórmula UCT, ver Seção 2.3);
4. Maior valor RAVE (valor da heurística RAVE (*Rapid Value Estimation*) que não é utilizada neste trabalho e não é discutida no texto).

Como é possível perceber, os procedimentos que desfazem as ações executadas no tabuleiro são possíveis devido à implementação de tabuleiro utilizada pelo SDM-Go e também presente no agente Fuego. Esta implementação é capaz de armazenar as peças capturadas e possibilita que as ações sejam desfeitas corretamente, retornando o tabuleiro ao estado inicial anterior a execução da sequência de ações. Os procedimentos *PercorreArvore*, *SimulaJogo* e *AtualizaArvore*, que implementam as fases de construção da árvore, *playout* e retropropagação, respectivamente, são detalhados em seguida. A Figura 4.5 apresenta o pseudocódigo do procedimento *PercorreArvore* que é responsável pela fase de construção da árvore da busca MCTS implementada pelo SDM-Go.

O procedimento *PercorreArvore* recebe a variável *EstadoFinal*, que tem como objetivo indicar se durante o caminho percorrido um estado de final de jogo foi alcançado. O procedimento também faz uso da variável global *ArvoreBusca* que permite acessar a árvore de busca que está sendo construída, inclusive de maneira paralela com outras threads. As instruções deste procedimento são comentadas a seguir:

Linha 1. O nó que representa o tabuleiro corrente é obtido através da raiz da árvore de busca.

Linha 2. A sequência de ações que formam o caminho percorrido na árvore é iniciada com o nó raiz.

Linha 3. A variável *booleana ContinuaCaminho*, que mantém o laço utilizado para percorrer um caminho pela árvore, é iniciada com o valor *VERDADEIRO*.

Linha 5. Verifica-se se o nó selecionado não possui filhos.

Linha 6. Caso o nó selecionado não possua filhos, as ações possíveis a partir deste nó são geradas através do procedimento *GeraAcoes* e armazenadas no vetor *Acoes*.

Linha 7. O teste verifica se existem ações disponíveis a partir do nó selecionado.

PercorreArvore(*EstadoFinal*)
Variável Global : *ArvoreBusca*

```

1: NoCorrente = Raiz(ArvoreBusca)
2: Inicia(CaminhoArvore, NoCorrente)
3: ContinuaCaminho = VERDADEIRO
4: while ContinuaCaminho do
5:   if !TemFilhos(NoCorrente) then
6:     Acoes = GeraAcoes(NoCorrente)
7:     if Vazio(Acoes) then
8:       EstadoFinal = VERDADEIRO
9:     else
10:      ExpandeNo(NoCorrente)
11:    end if
12:    ContinuaCaminho = FALSO
13:  end if
14:  NoCorrente = SelecionaFilho(NoCorrente)
15:  Adiciona(CaminhoArvore, NoCorrente)
16: end while
17: return CaminhoArvore

```

Figura 4.5: Pseudocódigo do procedimento da fase de construção da árvore da busca em árvore Monte-Carlo implementada pelo agente SDM-Go.

Linha 8. Caso não existam ações possíveis, isto significa que o final do jogo foi alcançado. Para indicar esta situação para o procedimento pai, *BuscaMCTS*, que invocou este, a variável *EstadoFinal* recebe o valor *VERDADEIRO*.

Linha 10. No caso em que existem ações legais a serem executadas, os nós filhos são gerados pela expansão do nó.

Linha 12. Após a expansão do nó (ou devido a se ter alcançado um estado final do jogo) a variável que controla o laço é ajustada para que este termine.

Linha 14. De maneira a aprofundar e continuar o caminho sendo percorrido na árvore, outro nó, filho do nó corrente, é escolhido segundo a política $\pi_{selecao}$ da fase de seleção da busca. Caso não existam nós filhos, devido à uma situação de término do jogo, o nó selecionado corresponde a ação *NULO*.

Linha 15. O nó filho selecionado é acrescentado à sequência de ações que formam o caminho percorrido na árvore.

Linha 17. Ao final do procedimento, o caminho formado pela sequência de nós selecionadas, contida na variável *CaminhoArvore*, é retornado.

É interessante ressaltar que no procedimento *SelecionaFilho*, na linha 14, caso a cor do nó filho seja a mesma da raiz, o valor é invertido, de maneira a adequar-se à natureza mini-max da árvore. Além disso, a sequência de ações retornada pelo procedimento *PercorreArvore* sempre contém na última posição o nó que foi recém adicionado à árvore. A única exceção acontece na situação em que um estado de final de jogo foi atingido. Desta

maneira, o procedimento *SimulaJogo*, executado em seguida, pode identificar facilmente a partir de qual nó realizar a simulação do jogo. Além disto, na implementação da busca utilizada pelo agente, o código que executa a expansão do nó, e consequentemente insere os filhos na árvore, apenas o faz quando o nó já foi visitado pelo menos 3 vezes desde a primeira vez que a busca foi executada. Uma vez atingida esta marca, o nó que é selecionado para ser expandido tem todos seus filhos acrescentados à árvore. A Figura 4.6 a seguir, apresenta o pseudocódigo do procedimento que implementa a fase de *playout*.

SimulacaoJogo(*SeqAcoes*) :

```

1: NoCorrente = Ultimo(SeqAcoes)
2: Continua = VERDADEIRO
3: while Continua do
4:   NoCorrente = SimulaAcao(NoCorrente)
5:   if AcaoNulo(NoCorrente) then
6:     Continua = FALSO
7:   end if
8:   Adiciona(SeqAcoes, NoCorrente)
9: end while

```

Figura 4.6: Pseudocódigo do procedimento de simulação de jogo da busca em árvore Monte-Carlo implementada pelo agente SDM-Go.

O procedimento *SimulaJogo* recebe como parâmetro a sequência de nós que formam o caminho percorrido na árvore durante a fase de construção da árvore. O objetivo é simular jogadas de maneira a rapidamente alcançar o final do jogo. Cada linha do pseudocódigo é comentada a seguir:

Linha 1. O último nó do caminho percorrido anteriormente é obtido da sequência recebida como parâmetro.

Linha 4. Uma ação disponível a partir do nó atual é gerada de acordo com a política *self-play* π_{playout} da fase de *playout* e o nó correspondente é construído.

Linha 5. Verifica-se se a ação gerada no passo anterior é a ação *NULO*.

Linha 6. A ação *NULO* indica que o término do jogo foi atingido e portanto o laço de repetição do procedimento deve terminar, através da atribuição do valor *FALSO* à variável de controle *Continua*.

Linha 8. O nó correspondente a ação gerada no passo da linha 4 é acrescentado à sequência de nós que formam o episódio corrente da busca.

Conforme visto no pseudocódigo do procedimento *SimulaJogo*, as ações da fase de *playout* são continuamente geradas até a ação *NULO*. A escolha destas ações geradas na linha 4 é feita de acordo com regras que definem uma política π_{playout} que estabelece a seguinte ordem de prioridade (da maior para a menor):

1. *Nakade*;

2. ameaça de captura de peças adversárias (*atari capture*);
3. defesa contra captura de peças pelo adversário (*atari defense*);
4. redução de liberdades de um grupo inimigo;
5. movimento pertencente a algum padrão de peças (forma);
6. captura de peças adversárias;
7. movimento aleatório;
8. passar a vez;
9. movimento nulo;

As regras utilizadas pelo *playout* correspondem a presença de situações específicas no tabuleiro, discutidas anteriormente na Seção 2.1. Ao término do procedimento, quando um estado final do jogo for alcançado, a sequência de nós corresponde às ações que, escolhidas a partir da árvore, e em seguida a partir da simulação de jogadas, representam uma partida completa do jogo. Após obter o resultado do estado final do tabuleiro em um dado episódio de simulação da busca, os nós da árvore que foram selecionados no caminho percorrido terão seus valores atualizados, conforme definido pela equação 2.4 apresentada na Seção 2.3. Convém salientar que a busca do SDM-Go difere daquela do Fuego essencialmente no que diz respeito ao termo $Q^t(s, a)$ da equação 2.4 que representa a qualidade do nó. No Fuego, o valor do termo de qualidade $Q(s_i, a)$ de um nó s_i que tenha sido inserido na árvore durante a fase de construção da árvore do episódio corrente é definido pela abordagem *Prior Knowledge*, mesmo que s_i já pertença à árvore. Este é mais um dos aspectos supervisionados do jogador Fuego, além das simulações. Conforme apresentado na próxima Seção, o SDM-Go, com o intuito de reduzir tal supervisão, investiga a substituição do valor $Q^t(s, a)$ obtido por *Prior Knowledge* pelo valor armazenado na memória SDM. Esta abordagem adotada pelo SDM-Go introduz o uso do histórico das avaliações no processo de aprendizagem, que é desprezado pelo agente Fuego.

O procedimento *AtualizaArvore*, cujo pseudocódigo é mostrado a seguir, na Figura 4.7, realiza esta tarefa, implementando a fase de retropropagação da busca. Ao avaliar o tabuleiro final do jogo simulado no episódio da busca, o resultado é obtido considerando a ótica do jogador que está realizando a busca, ou seja, o valor calculado representa vitória ou derrota para o jogador que tem a vez de jogar. Por exemplo, caso seja o jogador das peças pretas que está realizando a busca e, após a avaliação do tabuleiro final, calcula-se que este vence a partida, o valor do resultado é 1. Porém, sob a ótica do oponente, ocorreu uma derrota e para ele o valor do resultado, adequadamente refletindo este fato, é 0. Assim, a retropropagação do resultado da simulação é feita considerando a natureza mini-max da árvore de busca. O resultado é invertido ao atualizar-se os valores dos nós que representam situações de tabuleiro a partir das quais é a vez do oponente jogar.

AtualizaArvore(*Resultado*, *SeqAcoes*) :

```

1: ResultadoInverso = Inverte(Resultado)
2: MinMax = 0
3: for all No ∈ SeqAcoes do
4:   if NumeroPar(MinMax) then
5:     Atualiza(No, Resultado)
6:   else
7:     Atualiza(No, ResultadoInverso)
8:   end if
9:   MinMax = MinMax + 1
10: end for

```

Figura 4.7: Pseudocódigo do procedimento de atualização dos nós percorridos na árvore durante um episódio da busca implementada pelo agente SDM-Go.

Cada linha do pseudocódigo do procedimento *AtualizaArvore* é comentada a seguir:

Linha 1. O resultado obtido do tabuleiro final do episódio é invertido. A inversão é necessária para adequar-se à natureza mini-max da árvore de busca.

Linha 2. Para saber o momento correto de utilizar o resultado invertido, a variável *MinMax* é iniciada com o valor zero. Sempre que a variável armazena um valor par, a atualização é feita utilizando o resultado conforme obtido originalmente, enquanto que o resultado invertido é utilizando sempre que o valor armazenado na variável é ímpar.

Linha 4. O teste verifica se o valor da variável *MinMax* é par, com o objetivo de determinar qual valor de resultado deve ser utilizado para atualizar o nó a seguir.

Linha 5. O nó é atualizado com o valor de resultado originalmente obtido do tabuleiro final do episódio.

Linha 7. O nó é atualizado com o valor de resultado invertido.

Linha 9. Antes do início da próxima iteração, para atualizar o próximo nó, a variável *MinMax* tem seu valor incrementado, refletindo a mudança da natureza mini-max dos nós a medida que aprofunda-se no caminho do caminho percorrido na árvore.

Assim, o procedimento *AtualizaArvore*, levando em consideração a natureza *mini-max* da árvore, consegue atualizar adequadamente os nós do caminho percorrido, invertendo o resultado final sempre que o nó em questão é um nó *min*, ou seja, um nó que corresponde a uma situação onde o oponente realiza a jogada. Este comportamento da implementação do agente SDM-Go impacta diretamente na escolha dos nós filhos durante a fase de seleção. No procedimento *PercorreArvore*, mostrado anteriormente no pseudocódigo da Figura 4.5, ao invocar a rotina *SelecionaFilho* os valores dos nós filhos são devidamente invertidos, adequando-se corretamente à natureza *mini-max* da árvore.

4.1.1.1 Melhorando a Busca em Árvore Monte-Carlo com o Algoritmo UCT (*Upper Confidence Bound*)

Como no Fuego, a busca MCTS do SDM-Go pode ser melhorada pela utilização do algoritmo UCT (*Upper Confidence Bound*), que é implementado e executado durante a fase de seleção para encorajar movimentos exploratórios na árvore. Isto é, o algoritmo procura evitar que a escolha dos nós na fase de seleção seja dominada apenas por aqueles que possuem os maiores valores, possibilitando que nós que foram pouco visitados durante a busca sejam também escolhidos. Este objetivo é alcançado através da equação 2.4, apresentada na Seção 2.3 e utilizada pelo algoritmo UCT para balancear movimentos que exploram nós desconhecidas da árvore e movimentos que exploram nós cujos resultados são mais vantajosos. A introdução desta equação não altera o funcionamento geral da busca MCTS, que segue os mesmos passos descritos na Seção 2.3. O único impacto causado pela inserção da equação 2.4 diz respeito à política de seleção $\pi_{selecao}$ adotada durante a construção da árvore. Esta política é implementada no procedimento *SelecionaFilho*, apresentado anteriormente no pseudocódigo da Figura 4.5. A Figura 4.8 [Silver 2009] apresenta um exemplo do funcionamento da busca MCTS melhorada pelo algoritmo UCT.

A busca exemplificada pela Figura 4.8 [Gelly e Silver 2011] mostra 5 episódios de simulação a partir dos quais é possível observar a construção da árvore orientada pelo algoritmo UCT. A fase de construção da árvore é distinguida da fase de *playout* pela linha horizontal e as políticas utilizadas em cada fase são devidamente indicadas. A fase de retropropagação não é explicitamente demonstrada, mas pode ser observada através da mudança dos valores contidos nos nós entre o término de um episódio e o início de outro. Cada nó armazena dois valores, apresentados na Figura 4.8 no formato i/j , onde i representa o número de vitórias obtidas em episódios da busca corrente cujos caminho passaram por ele e j representa o número total de episódios da busca corrente cujos caminhos passaram por ele. É interessante ressaltar que no caso em que a sub-árvore recuperada da busca anterior (ver Seção 3.1), i e j sofrem o impacto dos valores de vitória e visitas, respectivamente, ocorridas nela. Os nós são ainda diferenciados de acordo com seu envolvimento em cada episódio da busca. Um nó recém acrescentado à árvore devido a fase de construção da árvore é representado na forma de uma estrela, enquanto outros nós que já pertenciam a árvore anteriormente são representados simplesmente por um círculo. Os nós percorridos durante a fase de *playout* e que, portanto, não são acrescentados à árvore são mostrados como losangos. Os nós terminais, que correspondem a tabuleiros finais de um jogo, são representados como quadrados. As cores preto e branco dos nós se alternam indicando qual jogador tem a vez de jogar a partir da situação de tabuleiro correspondente a cada nó. O caminho percorrido durante cada episódio é indicado pelas linhas verticais que terminam em um quadrado (estado final de jogo). A árvore apresentada no exemplo é binária, ou seja, cada nó tem exatamente dois filhos. O processo ilustrado pela Figura 4.8

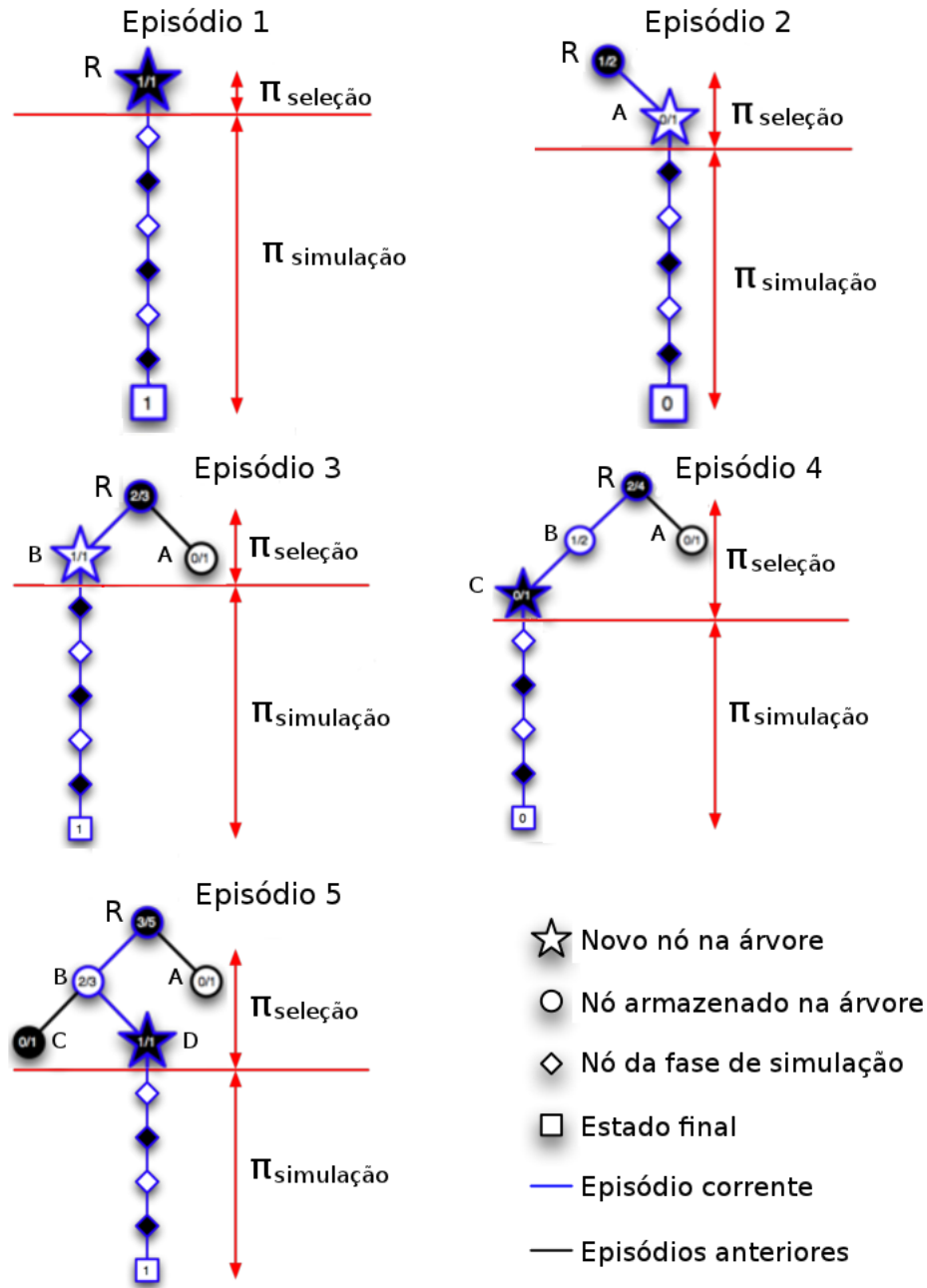


Figura 4.8: Ilustração da busca em árvore Monte-Carlo.

considera que, no início, a árvore está vazia. No primeiro episódio o nó raiz é acrescentado e, a partir dele, a fase de *playout* é executada pela primeira vez. No segundo episódio, a fase de seleção inicia no nó raiz e, como nem todos seus filhos foram visitados, um dos não visitados é escolhido aleatoriamente, uma vez que o SDM-Go não usa heurísticas. Sempre que algum nó não visitado é encontrado durante a fase de seleção, este é preferido em relação a outros nós do mesmo nível que já tenham sido visitados. Caso haja mais de um nó ainda não visitado, a escolha entre estes é feita aleatoriamente. O filho escolhido é o nó **A** que é acrescentado à árvore na fase de construção da árvore e a partir do qual a fase de *playout* se inicia. Após obter o resultado da simulação, os valores dos nós percorridos são atualizados, começando a partir do nó folha **A** em direção à raiz **R**:

1. O nó **A** tem a contagem de visitas durante a simulação incrementada, totalizando 1. O resultado obtido a partir do estado final desta simulação é derrota e, portanto, o número de vitórias associados ao nó é zero (0/1);
2. durante as simulações, o nó raiz **R** foi visitado duas vezes e em apenas uma o resultado final foi de vitória (1/2).

Convém salientar que, a título de simplificação, no exemplo da Figura 4.8 não estão representadas as inversões de valores das avaliações dos nós, em virtude do caráter *mini-max* da árvore de busca, que ocorrem durante a seleção e a retropropagação (ver seções 4.2.1 e 4.3.1). No terceiro episódio, o nó **B**, filho da raiz ainda não visitado, será escolhido, em detrimento do nó **A** já visitado, de acordo com o balanceamento almejado pelo algoritmo UCT. O resultado da simulação desta vez é vitória e os valores do nó raiz e do novo filho são atualizados na fase de retropropagação:

1. O nó **B** foi visitado apenas uma vez durante os episódios de simulação e obteve uma vitória (1/1);
2. o nó **R** aumenta para 3 o número de simulações nas quais participou e agora 2 vitórias foram obtidas nestes episódios (2/3).

No quarto episódio, os valores UCT calculados pela equação 2.4 para ambos os nós filhos da raiz serão comparados. De acordo com os valores que foram armazenados a partir dos episódios anteriores, a avaliação do nó **A** é $0.976853372C$, onde C é uma constante arbitrária que pondera o impacto do segundo termo da equação UCT, responsável por encorajar ações pouco simuladas. Como este valor é constante, a variável C será omitida daqui em diante no texto quando os valores calculados para os nós forem apresentados. O nó **B** é avaliado com o valor 1.976853372. Assim, a escolha do algoritmo UCT é pelo **B**, que possui o maior valor de avaliação. Cabe ressaltar que a diferença entre os valores de ambos os nós, **A** e **B**, é de 1. Ou seja, como ambos os nós têm o mesmo número de visitas durante as simulações, o nó escolhido é aquele que possui o maior número de vitórias, neste caso, **B**, que, com 1 vitória a mais, supera **A** exatamente por 1. Nenhum filho do nó **B** foi visitado ainda e, portanto, um deles é escolhido aleatoriamente e acrescentado à árvore na fase de construção da árvore. Este é o nó **C**, a partir do qual mais uma simulação é executada até que um estado final seja atingido. Após a retropropagação do resultado desta última simulação, os valores dos nós do caminho percorrido são:

1. O nó **C** recém adicionado possui uma visita durante as simulações e nenhuma vitória (0/1);
2. o nó **B** tem suas visitas aumentadas para 2 e permanece com apenas uma vitória durante as simulações (1/2);

3. o nó raiz **R** agora conta com 4 simulações que partiram dele e 2 vitórias alcançadas (2/4).

Durante o quinto episódio da busca, a fase de construção da árvore parte da raiz e escolhe o nó **B** novamente, pois sua avaliação de 1.775925249 supera a avaliação do nó **A** que é de 1.09732401. O nó filho de **B** ainda não visitado tem a preferência sobre **C** que já foi visitado uma vez durante as simulações. O nó **D** é adicionado e a partir dele a busca prossegue, executando a fase de *playout*. O resultado da simulação, vitória neste caso, é utilizado pela retropropagação para atualizar os nós do caminho percorrido na árvore:

1. o nó **D** foi visitado uma vez, neste episódio da busca e obteve uma vitória (1/1);
2. o nó **B** agora participou de 3 simulações e o número de vitórias obtidas nestas simulações é 2 (2/3);
3. o nó raiz **R** participou de todas as 5 simulações, nas quais 4 terminaram em vitória (4/5).

Ao final do quinto episódio, os nós da árvore possuem as seguintes avaliações, conforme calculadas pela equação 2.4 do algoritmo UCT:

1. Nó **A**: 1.182345131
2. Nó **B**: 2.68262728
3. Nó **C**: 0.976853372
4. Nó **D**: 1.976853372

Assim, verifica-se que a árvore de busca cresce com a adição de um nó a cada episódio e os valores dos nós são gradativamente atualizados. A estratégia implementada através do algoritmo UCT encoraja a exploração de nós ainda não visitados, ao mesmo tempo, dedica atenção também aos nós cujos valores se apresentam mais promissores. Ao final da busca, a ação associada ao melhor nó filho da raiz será escolhida para ser executada no jogo pelo agente. Conforme explicado anteriormente, o SDM-Go permite quatro critérios diferentes para escolher o *melhor* movimento. O critério mais utilizado e que apresenta melhores resultados é o de escolher a ação que possui maior taxa de vitórias [Silver 2009]. No exemplo de busca apresentado na Figura 4.8, a ação escolhida após os cinco episódios de simulação seria aquela correspondente ao nó **B**, que possui a maior taxa de vitórias entre os nós filhos da raiz (2/3).

4.1.1.2 Uso da Memória Esparsamente Distribuída na Busca do SDM-Go

No SDM-Go, alguns estados de tabuleiros já visitados em outros jogos e os valores a eles associados são armazenados na memória esparsamente distribuída. Posteriormente,

durante um novo jogo, os valores destes estados armazenados são empregados para avaliar os nós gerados durante a fase de construção da árvore. O tabuleiro resultante da ação associada a cada novo nó s_i , gerado na expansão, é comparado com os tabuleiros previamente memorizados, também referidos como **protótipos**. Aqueles tabuleiros ou protótipos P_j que atendem a um nível mínimo de similaridade com o tabuleiro associado a ação de s_i são selecionados. O valor de cada protótipo P_j é empregado, de forma ponderada pela similaridade, para fornecer uma avaliação prévia do nó s_i gerado na fase de construção da árvore. Depois deste processo, a busca segue normalmente conforme descrito anteriormente. Entretanto, durante a fase de retropropagação, o agente SDM-Go aproveita o resultado obtido ao final da simulação para, além de atualizar os valores dos nós do caminho percorrido na árvore, também atualizar os valores dos protótipos que são similares a estes nós. Para viabilizar a aplicação desta abordagem pelo agente SDM-Go, é necessária a comunicação do módulo de busca MCTS com os outros dois módulos presentes na arquitetura. Os processos de avaliação prévia dos nós e atualização dos valores dos protótipos são explicados em maiores detalhes nas seções 4.2 e 4.3, respectivamente. Porém, antes é necessário entender os dois outros módulos da arquitetura do SDM-Go, o de representação do tabuleiro em vetores de bits (*Bit Board Representation - BBR*) e o da memória esparsamente distribuída (*Sparse Distributed Memory - SDM*), explicados a seguir nas seções 4.1.2 e 4.1.4.

4.1.2 Módulo de Representação em Bits do Tabuleiro (BBR)

Este módulo implementa a representação utilizada pelo agente SDM-Go para armazenar estados de tabuleiro na memória esparsamente distribuída. O objetivo deste módulo é tornar a representação de tabuleiro mais apropriada ao uso da memória SDM, acelerando o processo de acesso à memória. Isto se torna necessário devido ao fato de que a representação complexa adotada pelo agente Fuego (ver Seção 3.1) inviabiliza o uso da memória SDM. No jogo de Go, apesar de existirem tabuleiros menores, com tamanhos 5X5 ou 7X7, normalmente, os jogos ocorrem em tabuleiros contendo de 81 (9X9) até 361 intersecções (19X19). Cada intersecção ou ponto do tabuleiro pode estar em um de três estados distintos:

1. ocupado por uma peça preta;
2. ocupado por uma peça branca ou;
3. vazio.

Disto decorre que no maior tamanho de tabuleiro, 19X19, por exemplo, o espaço de estados atinge 10^{160} elementos, conforme apresenta a Tabela 1.1, mostrada no capítulo 1. Devido a este grande espaço de estados possíveis do jogo, a memória do agente SDM-Go deve estar apta a armazenar um elevado número de tabuleiros e, ao mesmo tempo, fazer

um bom uso da memória física disponível de maneira eficiente e econômica. Além disto, o uso da memória SDM deve interferir o mínimo possível no tempo de processamento da busca, para permitir que o maior número de episódios de simulação possa ser executado. Neste sentido, a representação empregada pelo agente SDM-Go deve reduzir o espaço requerido para armazenar um estado de tabuleiro e também tornar eficientes as operações envolvendo vários estados. A solução para atender a estes requisitos, armazenamento e processamento, é implementada na forma de um vetor de bits. A representação de tabuleiro usada pelo agente Fuego, que é empregada pelo módulo de busca MCTS, é transformada em um vetor de bits ao passar pelo módulo de representação em bits. A conversão da representação utilizada na busca para a representação em bits acontece durante a fase de construção da árvore, a fim de que seja possível fazer a comparação entre os tabuleiros correspondentes aos nós expandidos e os tabuleiros armazenados na memória SDM. Resumindo, o uso desta nova representação, baseada em vetores de bits, tem dois objetivos principais:

1. A utilização eficiente da memória física disponível, pois a representação baseada em vetores de bits necessita de apenas 1 bit para indicar a presença de uma peça no tabuleiro;
2. Processamento rápido e eficiente nas operações de comparação entre os tabuleiros correspondentes aos nós gerados na fase de construção da árvore e os tabuleiros armazenados na memória esparsamente distribuída, já que as operações envolvendo bits (*bitwise*) são primitivas implementadas diretamente no processador.

4.1.2.1 Representação de Tabuleiro em Vetor de Bits

Cada ponto no tabuleiro é representado por uma posição em um vetor de bits. Os valores binários 0 e 1 de cada posição indicam a ausência ou presença de uma peça, respectivamente. Em outras palavras, se um ponto vazio do tabuleiro é ocupado por uma peça, a posição correspondente no vetor tem o bit ligado, ou seja, seu valor é alterado para 1. De maneira complementar, se a peça que ocupa um ponto do tabuleiro é capturada, a posição do vetor é desligada, seu o valor é alterado para 0. O funcionamento do vetor de bits, conforme descrito até agora, permite representar quais pontos do tabuleiro estão ocupados e quais estão vazios. Para um tabuleiro de tamanho $N \times N$, existem $M = N^2$ intersecções ou pontos possíveis. Assim, o vetor de bits necessitaria de M posições para representar se cada ponto está ou não ocupado. Entretanto, ainda é necessário representar qual a cor da peça em cada ponto ocupado no tabuleiro. Portanto, mais um bit é necessário para armazenar esta informação e então para cada ponto do tabuleiro são necessários dois bits: um bit para indicar a presença de uma peça e outro bit para indicar a cor da peça. Desta maneira, a representação de um tabuleiro com M posições agora exige um vetor de $2M$ bits. As M primeiras posições do vetor indicam a presença das peças pretas, enquanto

as M posições restantes indicam a presença das peças brancas. Naturalmente, um ponto do tabuleiro não pode ser ocupado simultaneamente por duas peças, sejam elas de mesma cor ou de cores diferentes. Se o bit na posição i do vetor, pertencente à parte reservada para as peças pretas, está ligado, complementarmente o bit na posição $j = i + M$, na parte reservada para as peças brancas, está desligado. Por fim, caso nenhuma peça esteja ocupando o ponto no tabuleiro, tanto o bit na posição i quanto o bit na contraparte j estão desligados. A Figura 4.9 a seguir ilustra o vetor de bits de um tabuleiro 3X3.

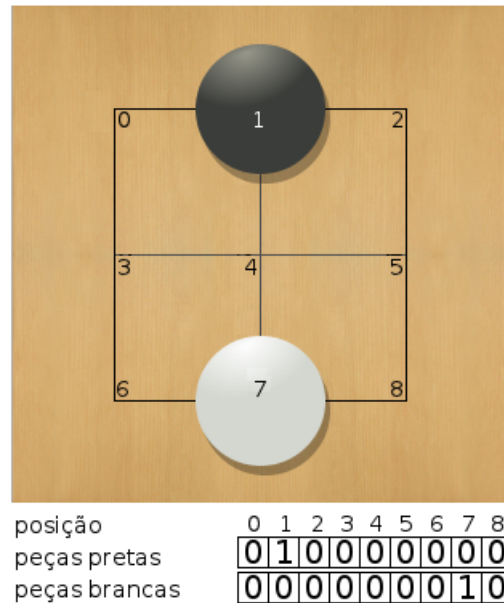


Figura 4.9: Representação de tabuleiro 3X3 usando um vetor de bits.

Os pontos do tabuleiro 3X3 mostrado na Figura 4.9 são enumerados, da esquerda para a direita, de cima para baixo. Apenas duas peças ocupam pontos deste tabuleiro: uma peça preta no ponto 1 e uma peça branca no ponto 7. O vetor de bits para um tabuleiro 3X3, que possui 9 intersecções ou pontos, necessita de 18 posições, de acordo com o que foi explicado anteriormente. A implementação do vetor de bits do SDM-Go é unidimensional, entretanto, na Figura 4.9 ele é mostrado bidimensionalmente para facilitar a visualização e o entendimento. As primeiras 9 posições indicam a presença das peças pretas e portanto apenas o bit na segunda posição (índice 1 do vetor) está ligado, correspondendo exatamente a peça preta na posição 1 enumerada no tabuleiro. Similarmente, nas 9 últimas posições que indicam a posição das peças brancas, apenas o bit na posição de índice 7 está ligado, o que corresponde a presença da peça branca no ponto enumerado 7 no tabuleiro. Assim é possível armazenar todas as informações necessárias para o funcionamento da memória esparsamente distribuída: a posição das peças no tabuleiro e a cor de cada peça. E ainda, como mencionado antes, esta representação possui a vantagem de ser a mais compacta possível e permitir que as operações de comparação sejam executadas de maneira simples e rápida, utilizando apenas operações *bitwise*.

A estrutura de dados utilizada para implementar a representação em vetores de bits,

da maneira como foi descrita anteriormente é apresentada a seguir. Cada campo da estrutura é explicado em detalhes depois.

```

1 struct BitBoardRepresentation
2 {
3     char* bits;
4     unsigned short size;
5     unsigned short bits_on;
6 };

```

A *struct BitBoardRepresentation* é uma implementação na linguagem de programação *C++* e possui três membros:

1. **bits**: um ponteiro do tipo *char* que aponta para a área de memória onde os bits que representam as peças no tabuleiro estão armazenados;
2. **size**: o tamanho do vetor, em número de bits, é armazenado nesta variável;
3. **bits_on**: esta variável mantém a informação sobre o número de bits ligados no vetor.

O vetor de bits é armazenado na área apontada pelo campo *bits*, um ponteiro de *char*. O tipo *char* é utilizado por ser o tipo primitivo da linguagem *C++* que ocupa o menor tamanho na memória física, 1 byte. O ideal em termos de aproveitamento da memória seria poder especificar uma variável que ocupasse exatamente 1 bit, pois assim o tamanho M do vetor de bits seria exatamente o necessário para a representação de um tabuleiro $N \times N$, onde $M = N^2$. Contudo, em *C++* isto não é possível e a utilização de tipos de variáveis que ocupam mais de 1 bit leva a ociosidade de alguns bits do espaço de memória alocado. Isto é, mais espaço do que é necessário é reservado e os bits excedentes nunca são utilizados na representação do tabuleiro. A utilização do tipo *char*, em oposição à outros tipos como *short int* e *int*, concorre para a minimização deste problema, como pode ser verificado pela comparação feita nas Tabelas 4.1, 4.2 e 4.3 para os tamanhos de tabuleiro 9x9, 13x13 e 19x19, respectivamente.

Tabela 4.1: Memória utilizada pelo vetor de bits implementado com base em vários tipos (Tabuleiro 9x9).

Tabuleiro 9x9 (162 bits necessários)			
Tipo	Tamanho do Tipo	Tamanho do Vetor	Bits Ociosos
char	1 byte	21 char (168 bits)	7 (4%)
short	2 bytes	11 short (176 bits)	14 (8%)
int	4 bytes	6 int (192 bits)	30 (16%)

4.1.2.2 Conversão do Tabuleiro da Busca para Vetor de Bits

Durante as fases de construção da árvore e retropropagação, os estados de tabuleiro correspondentes aos nós da árvore são transformados na representação de vetor de bits,

Tabela 4.2: Memória utilizada pelo vetor de bits implementado com base em vários tipos (Tabuleiro 13x13).

Tabuleiro 13x13 (338 bits necessários)			
Tipo	Tamanho do Tipo	Tamanho do Vetor	Bits Ociosos
char	1 byte	43 char (344 bits)	6 (2%)
short	2 bytes	22 short (352 bits)	14 (4%)
int	4 bytes	11 int (352 bits)	14 (4%)

Tabela 4.3: Memória utilizada pelo vetor de bits implementado com base em vários tipos (Tabuleiro 19x19).

Tabuleiro 19x19 (722 bits necessários)			
Tipo	Tamanho do Tipo	Tamanho do Vetor	Bits Ociosos
char	1 byte	91 char (728 bits)	6 (0.8%)
short	2 bytes	46 short (736 bits)	14 (2%)
int	4 bytes	23 int (736 bits)	13 (2%)

adequada às operações executadas pela memória SDM. Na fase de construção da árvore, o objetivo é obter uma avaliação prévia dos novos nós através da comparação com os tabuleiros armazenados na memória SDM. Já na retropropagação, o agente aproveita o resultado da simulação para, além de atualizar os valores dos nós do caminho percorrido, também atualizar os valores associados aos estados de tabuleiro memorizados. Em ambos os casos, primeiramente é necessário converter a representação de tabuleiro utilizada pela busca para a representação utilizada pela memória SDM. Isto é feito através do pseudocódigo apresentado na Figura 4.10 a seguir.

ConverteTabuleiro(*TabuleiroMCTS*) :

```

1: TamanhoVetor = 2 * Tamanho(TabuleiroMCTS) * Tamanho(TabuleiroMCTS)
2: VetorBits = CriaVetorBits(TamanhoVetor)
3: for all Ponto ∈ TabuleiroMCTS do
4:   if Ocupado(Ponto) then
5:     CorPeca = Cor(Ponto)
6:     PosicaoPeca = Posicao(Ponto)
7:     AdicionaPeca(VetorBits, CorPeca, PosicaoPeca)
8:   end if
9: end for
10: return VetorBits

```

Figura 4.10: Pseudocódigo do procedimento de conversão de tabuleiro usado pela busca MCTS para vetor de bits.

O procedimento *ConverteTabuleiro*, apresentado no pseudocódigo da Figura 4.10, inicia recebendo a representação de tabuleiro utilizado pela busca através da variável *TabuleiroMCTS*. Cada linha do pseudocódigo é comentada a seguir:

Linha 1. O tamanho necessário para o vetor de bits armazenar o estado de todas as posições do tabuleiro é calculado. A chamada *Tamanho*(*TabuleiroMCTS*) retorna o

tamanho do lado do tabuleiro. Portanto, para obter número de posições do tabuleiro é preciso calcular o quadrado do tamanho do tabuleiro. O vetor de bits necessita então do dobro de posições do tabuleiro.

Linha 2. O vetor de bits é criado com o tamanho especificado pelo parâmetro **TamanhoVetor**, calculado no passo anterior. O vetor, ao ser criado, é iniciado com o valor 0 para todas as posições, o que representa um tabuleiro totalmente vazio.

Linha 3. O laço itera sobre cada ponto do tabuleiro para obter a informação sobre seu estado.

Linha 4. O teste verifica se o ponto está ocupado por uma peça. Caso este teste falhe, ou seja, o ponto esteja vazio, nada necessita ser feito, pois na **linha 2** ao criar o vetor, todas as posições do vetor de bits já estão indicando vazio.

Linha 5. Primeiramente a cor da peça ocupando o ponto é obtida.

Linha 6. A posição enumerada correspondente ao ponto no tabuleiro é retornada pela chamada *Posicao(Ponto)*. A enumeração das posições do tabuleiro segue um esquema similar àquele mostrado anteriormente na Figura 4.9.

Linha 7. A chamada *AdicionaPeca(VetorBits, CorPeca, PosicaoPeca)* é responsável por encontrar a posição adequada no vetor de bits, segundo a cor da peça. O bit nesta posição é ligado (recebe o valor 1) para indicar a presença da peça.

Linha 10. Ao final da iteração sobre todos os pontos do tabuleiro, o vetor de bits está devidamente preenchido com a informação sobre as posições das peças e é então retornado.

A conversão do tabuleiro utilizado pela busca para a representação em bits facilita e agiliza a comunicação com a memória SDM. Assim, durante a fase de construção da árvore, é fácil obter a representação dos novos nós gerados, simplesmente alterando o vetor de bits correspondente ao nó expandido. Da mesma maneira, durante a fase de retropropagação, a representação dos nós do caminho percorrido na árvore é também facilmente obtida, alterando a representação em bits do nó inicial. Os procedimentos responsáveis por alterar os vetores de bits, adicionando e removendo peças, são explicados em detalhes na Seção 4.1.3 a seguir.

4.1.3 Atualização da Representação em Bits

Uma vez que o tabuleiro utilizado pela busca é convertido para a representação em bits, é possível representar as ações executadas através da alteração dos valores dos bits. Caso uma peça seja posicionada no tabuleiro, basta alterar o valor do bit correspondente no vetor para 1. Para desfazer a ação, ou seja, retirar a peça do tabuleiro, o valor do bit correspondente é alterado para 0. Os procedimentos para adicionar e remover peças são apresentados e discutidos a seguir. O procedimento *AdicionaPeca* atualiza o vetor de bits após a execução de uma ação, indicando adequadamente o posicionamento de uma peça

no tabuleiro. O pseudocódigo do algoritmo é apresentado na Figura 4.11. Cada linha do pseudocódigo é então comentada em seguida.

AdicionaPeca(*VetorBits*, *CorPeca*, *PosicaoPeca*) :

```

1: TamanhoByte = 8
2: if Cor(Branca) then
3:   PosicaoPeca = PosicaoPeca + InicioBranca
4: end if
5: PosicaoByte = PosicaoPeca / TamanhoByte
6: PosicaoBit = PosicaoPeca % TamanhoByte
7: LigaBit(VetorBits, PosicaoByte, PosicaoBit)

```

Figura 4.11: Pseudocódigo do procedimento para adicionar uma peça no tabuleiro representado pelo vetor de bits.

Linha 1. A variável *TamanhoByte* é uma constante que define o tamanho de um byte em número de bits.

Linha 2. O teste verifica se a cor da peça é branca. Caso a cor seja preta, nada necessita ser feito.

Linha 3. No vetor, os bits que indicam as peças brancas estão posicionados após todos os bits que indicam as peças pretas. Então é adicionado à posição recebida por parâmetro o *offset* a partir de onde começam os bits que representam as peças brancas.

Linha 5. A posição do byte que contém o bit a ser ligado é calculado.

Linha 6. A posição do bit dentro do byte é encontrada.

Linha 7. Basta agora ligar o bit do vetor com as devidas coordenadas calculadas na *linha 6*.

Este procedimento simples, apresentado na Figura 4.11, consegue rapidamente identificar o bit do vetor a ser atualizado e alterar seu valor. O cálculo das posições do bit a ser ligado e do byte no qual ele se encontra consiste apenas em obter o quociente e o resto de uma divisão. É simples entender o emprego da operação de divisão para encontrar estas posições de interesse. O vetor de bits é formado por uma sequência ou *array* do tipo char, conforme descrito antes na Seção 4.1.2.1. Isto é, cada posição do vetor, acessível através da linguagem de programação C++ utilizada na implementação do agente, corresponde a 1 byte na memória. Assim, é necessário determinar primeiramente em qual byte do vetor está o bit na posição *N* desejada. Isto é feito obtendo-se o quociente da divisão inteira executada na linha 5 do procedimento *AdicionaPeca*. A divisão pode ser vista como a pergunta **Quantos blocos de 8 bits, ou simplesmente bytes, são necessários para obter-se o total de N bits, ou o menor valor mais próximo de N?** Caso *N* seja múltiplo de 8, o quociente *Q* da divisão contém o valor exato que satisfaz a pergunta. Neste caso, o resto *R* da divisão é zero e isto significa que o primeiro bit (posição 0) do byte *Q* do vetor é aquele que deve ser alterado. Em caso contrário, se *N* não é múltiplo

de 8, o valor do quociente Q corresponde ao menor valor mais próximo de N mencionado na pergunta. É necessário então somar o valor do resto R ao valor de Q para obter N , o que significa que o bit desejado está na posição R do byte Q do vetor. A Figura 4.12 a seguir ilustra um exemplo do processo descrito.

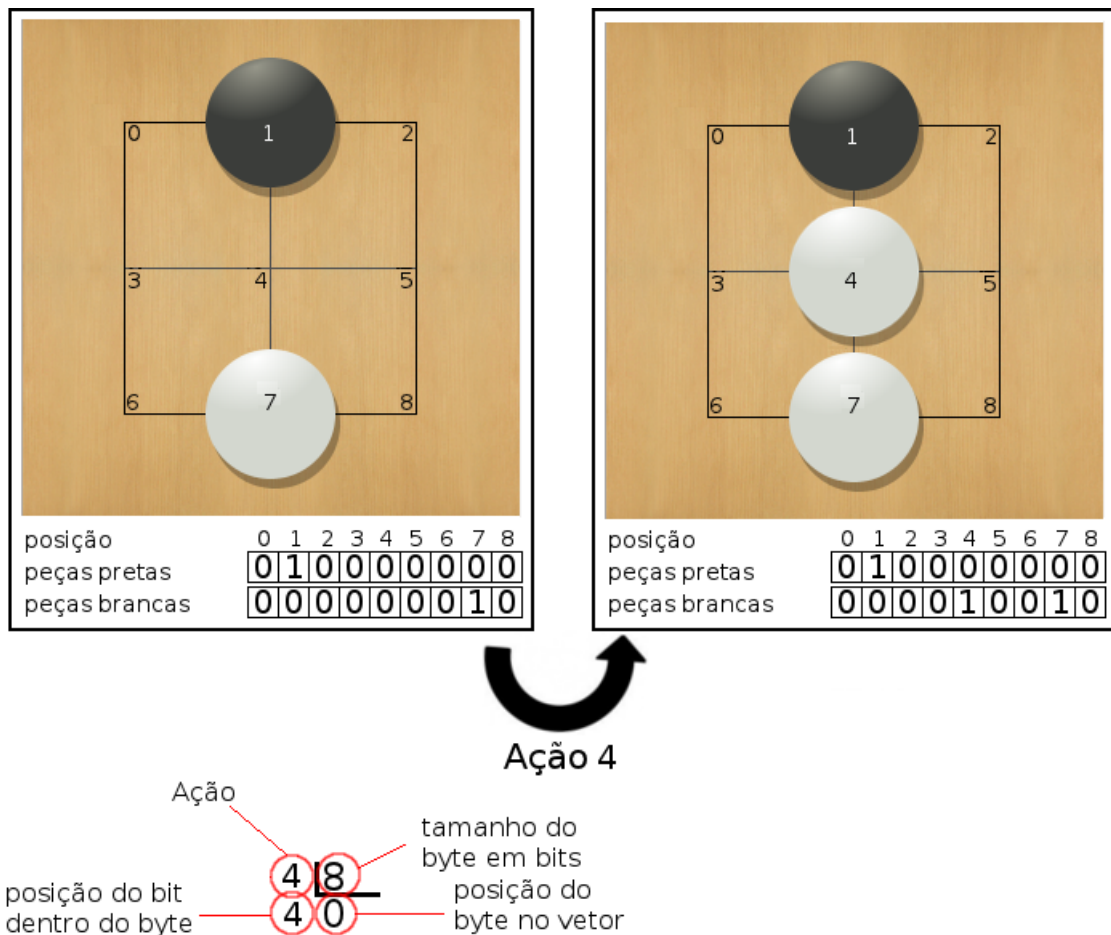


Figura 4.12: Exemplo do cálculo do bit a ser atualizado após a execução de uma ação.

No tabuleiro 3X3, apresentado na Figura 4.12, uma peça branca é colocada no ponto 4. O cálculo do bit a ser atualizado de maneira a refletir o estado resultante da execução da ação é feito conforme explicado antes. A ação contém a informação da posição do tabuleiro na qual a peça é posicionada, neste caso, 4, cujo valor é então dividido pelo tamanho em bits de um byte, 8. O quociente da divisão inteira indica qual byte do vetor contém o bit correspondente à ação. No exemplo da Figura, o valor 0 indica o primeiro byte do vetor. A posição do bit dentro do byte é dada pelo resto da divisão, cujo valor é, também, 4. Assim, o 4º bit do primeiro byte (byte 0) da porção do vetor correspondente às peças brancas é atualizado para representar o tabuleiro resultante após posicionada a peça branca, de acordo com o exemplo da Figura 4.12. Como a implementação do SDM-Go utiliza um vetor de bits unidimensional (conforme mencionado anteriormente, a representação da Figura tem caráter apenas didático), no exemplo da Figura 4.12, a posição calculada deve ser ajustada, levando em consideração que a ação executada

envolve uma peça da cor branca. Na implementação unidimensional do vetor de bits, as últimas posições são reservadas para as peças brancas, o que torna necessário adicionar um *offset* ao valor calculado. Este *offset* corresponde ao tamanho da primeira porção do vetor, reservado para as peças pretas, cujo valor é 9. Assim, considerando um vetor de bits unidimensional de tamanho 18, a posição do bit desejado, de valor 4 conforme calculado pela divisão, é somada com o *offset* 9, o que resulta no valor 13. Observe que esta posição corresponde àquela alterada na Figura 4.12, bastando considerar a primeira posição das peças brancas no vetor como a posição 9, dando sequência às posições enumeradas para a primeira parte do vetor. Para desfazer ações ou remover peças que foram, um processo semelhante ao apresentado na Figura 4.11 é utilizado. De fato, a única diferença reside na linha 7, a última do procedimento, onde um bit é desligado.

RemovePeca(*VetorBits*, *CorPeca*, *PosicaoPeca*) :

```

1: TamanhoByte = 8
2: if Cor(Branca) then
3:   PosicaoPeca = PosicaoPeca + InicioBrancas
4: end if
5: PosicaoByte = PosicaoPeca/TamanhoByte
6: PosicaoBit = PosicaoPeca%TamanhoByte
7: DesligaBit(VetorBits, PosicaoByte, PosicaoBit)
```

Figura 4.13: Pseudocódigo do procedimento para adicionar uma peça no tabuleiro representado pelo vetor de bits.

Uma vez calculada a posição do bit desejado, o passo seguinte no algoritmo mostrado na Figura 4.11 é ligar o bit, ou seja, atribuir-lhe o valor 1. No caso do algoritmo da Figura 4.13, o bit é desligado (valor 0). As atualizações nos valores dos bits são feitas de maneira bastante rápida, utilizando operações lógicas simples. A Seção 4.1.3.2 discute em detalhes a maneira como estas operações são empregadas para tratar os vetores de bits. A utilização dos procedimentos *AdicionaPeca* e *RemovePeca* permite que o vetor de bits correspondente a um estado de tabuleiro possa ser alterado, refletindo a execução de ações e também a reversão das mesmas. Desta maneira, durante a fase de construção da árvore, apenas a representação do nó expandido é construída. A partir do vetor de bits gerado, os nós filhos expandidos são obtidos através da alteração dos bits correspondentes as ações que geram cada filho. Similarmente, durante a retropropagação, apenas a representação do nó raiz é gerada e a partir desta, todos os demais nós do caminho percorrido são obtidos pela atualização dos bits do vetor. Em ambos os casos, uma vez que o vetor de bits do nó inicial é gerado, apenas é necessário que o módulo de busca informe as ações correspondentes aos demais nós que seguem. Assim, o SDM-Go mantém uma comunicação simples e eficiente entre os módulos de busca e de memória, minimizando a necessidade de troca de informações. O processamento do SDM-Go pode então prosseguir para executar os testes de similaridade. Os testes determinam o grau de similaridade

entre os nós da árvore de busca e os protótipos armazenados na memória SDM. No caso da fase de construção da árvore, os protótipos mais similares e seus respectivos valores são utilizados para computar uma avaliação prévia dos nós a serem selecionados. Na fase de retropropagação, juntamente com a atualização dos valores dos nós do caminho percorrida na árvore, os valores daqueles protótipos mais similares a estes nós são também atualizados. A Seção 4.1.3.1 a seguir explica em detalhes como os testes de similaridade são executados.

4.1.3.1 Cálculo de Similaridade entre Tabuleiros

A principal ideia empregada na abordagem do SDM-Go consiste em avaliar estados de tabuleiro desconhecidos a partir de estados similares sobre os quais o agente já possui alguma informação. O grau de similaridade entre os tabuleiros determina de quais protótipos será extraída a informação para avaliar os nós ainda desconhecidos, gerados durante a busca na fase de construção da árvore. A semelhança entre os tabuleiros também é utilizada na fase de retropropagação, para atualizar o valor dos protótipos armazenados, aproveitando o resultado de um episódio de simulação. Conforme explicado nas seções anteriores deste capítulo, a representação em bits permite que os testes de similaridade entre dois tabuleiros sejam executados de maneira bastante rápida, utilizando operações *bitwise*. A distância de Hamming é uma métrica que calcula a diferença entre duas sequências de acordo com o número de bits diferentes que elas possuem. Considere, por exemplo, dois vetores de bits, $B1$ e $B2$ e o conjunto de pares de bits $b1_i$ e $b2_i$ que ocupam a mesma posição i em cada um dos vetores $B1$ e $B2$ respectivamente. Para todas as posições i , a distância de Hamming calcula o número de vezes em que os bits $b1_i$ e $b2_i$ possuem valores diferentes. Esta é portanto, uma métrica apropriada para medir a similaridade entre os nós da árvore e os protótipos da memória, considerando que estes estão representados por vetores de bits (limitações desta métrica são apresentadas na seção 4.4). Além disto, a distância de Hamming pode ser facilmente calculada utilizando as operações *bitwise* empregadas nos demais procedimentos envolvendo os vetores de bits. A Figura 4.14 ilustra dois vetores de bits, $B1$ e $B2$, que representam dois estados de tabuleiro distintos.



Figura 4.14: Distância de Hamming entre dois vetores de bits diferentes.

Na Figura 4.14 os 6 bits de mesma posição cujos valores são diferentes estão destacados. A distância de Hamming entre os dois vetores $VB1$ e $VB2$ equivale exatamente a este número de bits diferentes. O pseudocódigo da Figura 4.15 a seguir apresenta o algoritmo

utilizado pelo SDM-Go para calcular o grau de similaridade entre dois vetores de bits através da distância de Hamming entre eles.

CalculaSimilaridade(*VetorBits1*, *VetorBits2*) :

```

1: DistanciaMaxima = Tamanho(VetorBits1)
2: Distancia = DistanciaHamming(VetorBits1, VetorBits2)
3: Similaridade = (DistanciaMaxima – Distancia)/DistanciaMaxima
4: return Similaridade

```

Figura 4.15: Pseudocódigo do procedimento que calcula o grau de similaridade entre dois vetores de bits.

Linha 1. A distância máxima possível entre dois vetores ocorre quando todos os bits diferem entre si.

Linha 2. A distância de Hamming entre os vetores de bits é calculada.

Linha 3. A similaridade é calculada em função da distância de Hamming. A diferença entre a distância de Hamming calculada e a distância máxima possível corresponde ao número de bits que são similares entre os dois vetores. Este valor é então dividido pelo tamanho total do vetor, que coincide com a distância máxima.

Linha 4. O procedimento termina, retornando o grau de similaridade calculado.

O grau de similaridade calculado na linha **3** do procedimento utiliza a distância de Hamming para subtrair seu valor da distância máxima possível e assim obter o número de bits similares entre os dois vetores. Uma vez calculado, este valor é dividido pelo tamanho do vetor, coincidente com o valor da distância máxima possível. Logo, o grau de similaridade representa a porcentagem de bits em cada vetor que possuem o mesmo valor. Caso todos os bits sejam diferentes, a distância de Hamming entre os vetores tem o mesmo valor da distância máxima e portanto o grau de similaridade calculado pela divisão na linha **3** é zero. Por outro lado, se todos os bits são idênticos, a distância de Hamming é zero e o grau de similaridade é 1 (100%). Nos demais casos, o resultado é um número entre 0 e 1, que representa o grau de similaridade como a porcentagem de bits com mesmo valor em ambos os vetores. Na linha **2** do pseudocódigo da Figura 4.15 a distância de Hamming é obtida a partir da chamada de outro procedimento. A implementação da rotina que calcula a distância de Hamming utiliza a operação lógica **ou exclusivo** (*XOR*) entre os bits dos vetores. Esta e outras operações utilizadas para manipular os vetores de bits são apresentada em detalhes na próxima Seção 4.1.3.2.

4.1.3.2 Operações *bitwise*

Um dos objetivos propostos pela representação do tabuleiro em bits é a rapidez das operações necessárias ao funcionamento do agente SDM-Go. As operações entre bits são implementadas diretamente no *hardware*, o que possibilita que sejam executadas mais

rapidamente do que em *software*. Estas operações formam a base dos cálculos executados sobre os vetores de bits gerados e manipulados durante a busca, assim como aqueles armazenados na memória. Tais cálculos envolvem, além da atualização da representação em bits para refletir as mudanças ocorridas no tabuleiro, testes de comparação que visam obter o grau de similaridade entre dois ou mais vetores de bits. Para tanto, operações lógicas simples entre os bits, também chamadas de *bitwise*, são empregadas:

- conjunção (*AND*);
- disjunção (*OR*);
- negação (*NOT*);
- disjunção exclusiva (*XOR*) e;
- deslocamento (*shift*).

No pseudocódigo da Figura 4.11, apresentado anteriormente na Seção 4.1.2.2, na linha 7 o procedimento *LigaBit*(*VetorBits*, *PosicaoByte*, *PosicaoBit*) utiliza as operações *OR* e *SHIFT* para ligar o bit do vetor com as coordenadas do byte onde está contido e sua posição dentro deste. O pseudocódigo do procedimento é apresentado a seguir na Figura 4.16

LigaBit(*VetorBits*, *PosicaoByte*, *PosicaoBit*) :

```

1: Mascara = 1
2: ShiftEsquerda(Mascara, PosicaoBit)
3: VetorBits[PosicaoByte] = OR(VetorBits[PosicaoByte], Mascara)
```

Figura 4.16: Pseudocódigo do procedimento para ligar um bit.

A linguagem *C++* utilizada na implementação do SDM-Go não permite que bits sejam acessados individualmente. Portanto, as operações bitwise são executadas através da unidade mais básica acessível, um byte. Os valores binários 1 e 0 correspondem aos valores lógicos verdadeiro e falso, respectivamente. Cada linha do procedimento é comentada a seguir:

Linha 1. Conforme mencionado antes, não é possível alterar diretamente um bit apenas. Portanto uma máscara é usada para obter o resultado desejado após a operação *bitwise*. A máscara é iniciada com o valor 1.

Linha 2. Os bits da máscara são deslocados *n* vezes para a esquerda, onde *n* corresponde ao valor da posição do bit a ser ligado.

Linha 3. Após alterar adequadamente os bits da máscara, a operação lógica *OR* é executada entre todos os bits do byte da máscara e do vetor. O resultado é armazenado de volta no byte do vetor de bits.

O sucesso do procedimento *LigaBit* depende da configuração correta dos bits da máscara. É necessário utilizar uma máscara cujos bits tenham valores que permitam obter o resultado desejado ao final da operação lógica *OR*, que neste caso é o valor 1 atribuído ao bit b_i do vetor, onde i é a posição do bit dentro do byte. A operação *OR* possui duas propriedades interessantes que viabilizam o bom funcionamento do procedimento *LigaBit*:

1. a disjunção entre X e 1 sempre é 1, qualquer que seja o valor X ;
2. a disjunção entre X e 0 é sempre X , qualquer que seja o valor de X .

Assim, seja m_i o bit da máscara na posição i , para garantir que o resultado da disjunção $m_i \vee b_i$ seja 1, basta que o valor de m_i também seja 1. Por outro lado, para manter os valores dos outros bits inalterados, seja m_j qualquer outro bit da máscara diferente de m_i , basta que o valor de m_j seja 0. Levando em consideração este raciocínio, o byte da máscara é iniciado com o valor 1, o que corresponde a sequência de bits mostrada na Figura 4.17. Assim, todos os bits do byte têm o valor zero, a exceção do primeiro bit que tem valor 1. Observe que o endereço dos bits crescem da direita para a esquerda e o bit menos significativo é armazenado no menor endereço. Esta maneira de ler os bits e consequentemente o valor do byte é denominada *little-endian*. Uma vez que a máscara de bits é iniciada com apenas um bit de valor 1, este é deslocado i vezes para a esquerda de maneira a ocupar a mesma posição i do bit b_i que se deseja ligar. O resultado da operação *OR* altera apenas o valor de b_i e mantém os demais bits inalterados. A Figura 4.17 apresenta um exemplo onde o bit a ser ligado está na posição $i=4$.

Ligar o bit na posição $i = 4$							
	7	6	5	4	3	2	1 0
Vetor de bits	X	X	X	0	X	X	X X
	7	6	5	4	3	2	1 0
Máscara	0	0	0	0	0	0	0 1
	7	6	5	4	3	2	1 0
Máscara após 4 x shift para esquerda	0	0	0	1	0	0	0 0
	7	6	5	4	3	2	1 0
Operação OR (disjunção)	0	0	0	1	0	0	0 0
	X	X	X	0	X	X	X X
Resultado	X	X	X	1	X	X	X X

Figura 4.17: Exemplo do processo para ligar um bit do vetor.

O procedimento para desligar um bit é bastante similar ao de ligar. A diferença consiste na operação utilizada para desligar o bit. Enquanto que no procedimento *LigarBit*, a operação *OR* é empregada, no procedimento *DesligaBit*, apresentado na Figura 4.18, as operações *AND* e *NOT* são usadas. Cada linha do pseudocódigo do procedimento

DesligaBit(*VetorBits*, *PosicaoByte*, *PosicaoBit*) :

```

1: Mascara = 1
2: ShiftEsquerda(Mascara, PosicaoBit)
3: Mascara = NOT(Mascara)
4: VetorBits[PosicaoByte] = AND(VetorBits[PosicaoByte], Mascara)

```

Figura 4.18: Pseudocódigo do procedimento para desligar um bit.

DesligaBit é explicada em seguida.

Linha 1. Da mesma maneira que no procedimento *LigaBit*, uma máscara é utilizada e seu valor inicial é 1.

Linha 2. O bit 1 na primeira posição do byte da máscara é deslocado n vezes para a esquerda, onde n corresponde ao valor da posição do bit a ser desligado.

Linha 3. A máscara é alterada pela operação *NOT* que inverte os valores dos bits.

Linha 4. O resultado armazenado no vetor de bits é dado pela conjunção (*AND*) dos bits da máscara e os bits do vetor.

Da mesma maneira que o procedimento *LigaBit* se aproveita das propriedades da disjunção, o pseudocódigo apresentado na Figura 4.18 é construído para aproveitar duas propriedades inerentes da conjunção lógica (*AND*):

1. a conjunção entre um valor X e 0 é sempre 0, independente do valor de X ;
2. o resultado da conjunção é 1 se, e somente se, ambos os valores são também 1.

Portanto, para desligar o bit bi na posição i , o bit mi da máscara deve ser 0. Para obter exatamente o valor 0 em mi , os bits da máscara são deslocados i vezes para a esquerda, da mesma forma que anteriormente quando se desejava ligar o bit. Entretanto, agora, uma vez deslocados os bits, os valores são invertidos pela operação de negação (*NOT*) de maneira que os bits fiquem configurados como mostra a Figura 4.19. Assim, com mi assumindo o valor 0, basta executar a operação *AND* para garantir que o resultado seja um bit também com valor 0. Os demais bits da máscara, diferentes de mi , possuem todos o valor 1. Isto garante que os outros bits do vetor, diferentes de bi , mantenham seus valores inalterados. As duas propriedades da conjunção, mencionadas anteriormente, viabilizam o bom funcionamento do procedimento para desligar bits. A Figura 4.19 ilustra um exemplo onde se deseja desligar um bit do vetor.

Assim, na fase de construção da árvore, o tabuleiro correspondente ao nó expandido é convertido para sua representação na forma de vetor de bits, enquanto que na fase de retropropagação o tabuleiro convertido é aquele correspondente ao nó raiz. Em ambos os casos, a partir da representação em bits do tabuleiro é possível produzir a representação dos tabuleiros seguintes através das ações correspondentes aos nós subsequentes. No caso

Desligar o bit na posição $i = 4$	
Vetor de bits	7 6 5 4 3 2 1 0 X X X 1 X X X X
Máscara	7 6 5 4 3 2 1 0 0 0 0 0 0 0 0 1
Máscara após 4 x shift para esquerda	7 6 5 4 3 2 1 0 0 0 0 1 0 0 0 0
Máscara após Operação NOT	7 6 5 4 3 2 1 0 1 1 1 0 1 1 1 1
Operação AND (conjunção)	7 6 5 4 3 2 1 0 X X X 1 X X X X 1 1 1 0 1 1 1 1 $\wedge$
Resultado	7 6 5 4 3 2 1 0 X X X 0 X X X X

Figura 4.19: Exemplo do processo para desligar um bit do vetor.

da construção da árvore, cada nó filho do nó expandido produz um tabuleiro diferente, de acordo com a ação que representa. No caso da retropropagação, cada ação dos nós do caminho percorrido na árvore gera cada tabuleiro da sequência que leva ao estado terminal no final do episódio de simulação. Para obter o vetor de bits resultante da execução de cada ação, basta fazer a chamada do procedimento *AdicionaPeca*, de maneira similar ao que acontece no pseudocódigo da Figura 4.10. Isto é possível pois conforme descrito anteriormente na Seção 4.1.1 e ilustrado na Figura 4.2, as ações dos nós da árvore construída pela busca representam exatamente a posição na qual posicionar uma peça. A cor da peça é facilmente obtida considerando a natureza *mini-max* da árvore, em outras palavras, começando no nó raiz, cada nível de profundidade corresponde à cor oposta do nível anterior. Para desfazer as ações, uma abordagem semelhante àquela do procedimento *AdicionaPeca* é empregada e, de fato, a única diferença consiste em desligar o bit na posição do vetor correspondente a ação a ser desfeita. Desta maneira, é possível modificar adequadamente o vetor de bits para refletir a captura de peças e também para retornar ao estado original após a execução de uma sequência de ações.

A conversão do tabuleiro utilizado pela busca para o vetor de bits é importante para que o agente possa dar início à avaliação dos nós, no caso da fase de construção da árvore, ou à atualização dos protótipos da memória, no caso da fase de retropropagação. Em ambos os casos, é necessário calcular o grau de similaridade entre o vetor de bits correspondente ao tabuleiro e os vetores de bits correspondentes aos protótipos armazenados na memória SDM. Conforme explicado anteriormente na Seção 4.1.3.1, a similaridade entre vetores de bits é calculada através da distância de Hamming entre eles. No pseudocódigo apresentado na Figura 4.15, na linha **2**, o procedimento *DistanciaHamming* é responsável por calcular a distância entre dois vetores de bits. Este procedimento utiliza a operação

XOR para obter o número de bits que possuem valores diferentes, conforme mostrado no pseudocódigo da Figura 4.20, com cada linha comentada em seguida.

DistanciaHamming(*VetorBits1*, *VetorBits2*) :

```

1: Distancia = 0
2: Mascara = 1
3: for all Byte1 ∈ VetorBits1, Byte2 ∈ VetorBits2 do
4:   Resultado = XOR(Byte1, Byte2)
5:   for all i ∈ [1, TamanhoByte] do
6:     ValorBit = AND(Resultado, 1)
7:     Distancia = Distancia + ValorBit
8:     ShiftDireita(Resultado, 1)
9:   end for
10: end for
11: return Distancia
```

Figura 4.20: Pseudocódigo do procedimento para calcular a distância de Hamming.

- Linha 1.** A variável que armazena a distância calculada é iniciada com o valor zero.
- Linha 2.** Um byte de máscara é iniciado com o valor 1.
- Linha 3.** Cada par de bytes dos vetores de bits é iterado no laço.
- Linha 4.** A operação *XOR* é aplicada aos bytes de mesma posição dos vetores. O resultado é um byte onde em cada bit o valor 1 significa que os bits dos vetores são diferentes e o valor 0 que eles são idênticos.
- Linha 5.** Os 8 bits do byte *Resultado* são iterados para verificar o valor de cada um.
- Linha 6.** O valor do bit menos significativo é obtido através da operação lógica *AND* entre o byte *Resultado* e um byte cujo valor é 1.
- Linha 7.** O valor da distância de Hamming acumulado até agora é incrementado com o valor do bit, obtido no passo anterior.
- Linha 8.** Os bits do byte *Resultado* são deslocados uma posição para a direita. Desta maneira, o bit menos significativo é descartado e o bit seguinte toma seu lugar durante a próxima iteração do laço.
- Linha 11.** Após executar a operação *XOR* em todos os pares de bytes dos vetores, o valor calculado da distância de Hamming é retornado.

O procedimento que calcula a distância de Hamming acessa os vetores de bits byte a byte, pois, conforme mencionado anteriormente, esta é a menor unidade de memória acessível a partir da linguagem *C++*, na qual o SDM-Go está implementado. A operação lógica *XOR* é então aplicada a cada par de bytes *B1* e *B2* que ocupam uma mesma posição *i*. Todos os bits que contêm o valor de saída da operação *XOR* sobre os bits de *B1* e *B2* são armazenados em outro byte *B3*. O resultado da operação sobre cada par de bits *b1_j* e *b2_j* de mesma posição *j* é outro bit *b3_j*. A operação *XOR* possui uma propriedade

interessante que é aproveitada pelo SDM-Go para calcular a distância de Hamming entre os vetores de bits. Esta operação retorna o valor 1 se, e somente se, o bit $b1_j$ é diferente do bit $b2_j$. Em qualquer outro caso, o resultado da operação é 0. Assim, a distância de Hamming pode ser facilmente calculada como a soma dos valores dos bits resultantes da operação **XOR** entre cada par de bits de mesma posição j . Para efetuar esta soma, é necessário obter o valor individual de cada bit resultante $b3_j$, o que é feito através das operações *AND* e *Shift*. A máscara M que contém o valor 1 tem todos os bits com o valor 0 a exceção do bit menos significativo m_0 cujo valor é 1, conforme visto anteriormente nas Figuras 4.17 e 4.19. É importante lembrar também as duas propriedades da operação *AND*, mencionadas antes. Assim, seja X o valor do bit menos significativo $b3_0$ do byte $B3$ que contém o resultado da operação *XOR*, o resultado da operação *AND* entre $b3_0$ e m_0 , cujo valor é sempre 1, será:

- **0**, caso $X = 0$, pois a operação *AND* resulta sempre em 0 se um de seus operandos também é 0;
- **1**, caso $b3_0$ tenha o valor $X = 1$, pois o resultado da operação *AND* é 1 se, e somente se, ambos os operandos são também 1.

Como os demais bits m_i da máscara, com $i \neq 0$, possuem todos valor 0, o valor resultante da operação *AND* entre os bytes $B3$ e M é determinado apenas pelo bit menos significativo $b3_0$. Desta maneira, o valor de $b3_0$ é sempre refletido no resultado da operação *AND*. Uma vez que este valor é acumulado na variável *Distancia* na linha 8 do pseudocódigo da Figura 4.20, o deslocamento dos bits uma posição para a direita descarta $b3_0$ e o próximo bit toma seu lugar. O processo para obter o valor do bit menos significativo continua até que todos os 8 bits de R tenha sido processados. A Figura 4.21 ilustra este processo em maiores detalhes.

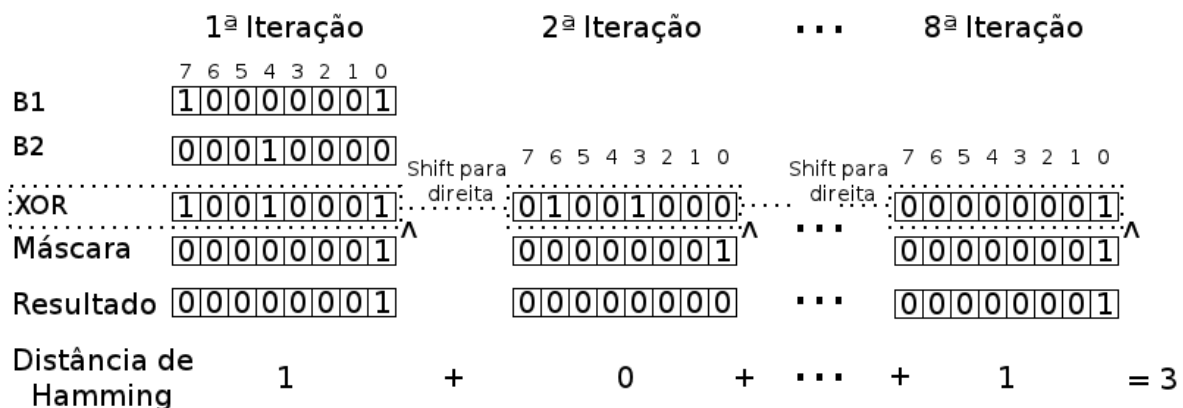


Figura 4.21: Ilustração do processo para calcular a distância de Hamming utilizando as operações *bitwise XOR* e *Shift*.

A distância de Hamming entre dois de bits, $B1$ e $B2$, é calculada na Figura 4.21 através das operações *bitwise XOR* e *Shift*, conforme explicado anteriormente. O processo é repetido para os 8 pares de bits dos bytes. Após obter o valor de cada bit do

resultado da operação *XOR*, mais um par de bytes é submetido ao mesmo processo até que a distância entre todos os bytes dos vetores tenha sido calculada. Os valores obtidos dos bits resultantes são acumulados durante todo o procedimento, como mostra o pseudocódigo da Figura 4.20, naturalmente resultando ao final na distância de Hamming entre os dois vetores de bits. As operações bitwise permitem a utilização dos vetores de bits como representação de estados de tabuleiro da maneira mais rápida possível, minimizando o tempo de processamento destes durante o funcionamento do SDM-Go. Esta característica e a eficiência na utilização da memória física, devido à baixa quantidade de espaço requerida pelos vetores de bits, são fundamentais para o funcionamento do SDM-Go. Desta maneira, a memória SDM se torna uma alternativa viável ao processo de avaliação prévia dos nós durante a fase de construção da árvore. A contínua atualização da memória durante os episódios de busca, na fase de retropropagação, também é possível devido a estas características vantajosas dos vetores de bits. O módulo da memória SDM é abordado em detalhes a seguir na Seção 4.1.4.

4.1.4 Módulo da Memória Esparsamente Distribuída

Neste módulo encontra-se a implementação da memória esparsamente distribuída (SDM) utilizada pelo SDM-Go. Durante os episódios da busca MCTS, alguns estados de tabuleiro são selecionados e armazenados na memória juntamente com seus valores associados. Todos os estados presentes na memória são convertidos em vetores de bits pelo módulo de representação em bits, descrito anteriormente na Seção 4.1.2. Uma vez armazenados na memória, os vetores de bits que representam os estados de tabuleiro são chamados também de **protótipos**. Durante a fase de construção da árvore, O SDM-Go utiliza os protótipos para computar uma avaliação prévia para os novos nós gerados, ao contrário das heurísticas codificadas por especialistas humanos utilizadas pelo agente Fuego. Na retropropagação, o resultado da simulação, além de atualizar os nós do caminho percorrido na árvore, como acontece na busca MCTS do agente Fuego, também é utilizado pelo SDM-Go para atualizar os valores associados aos protótipos armazenados na memória.

O modelo de memória SDM, apresentado na Seção 2.4 pressupõe a existência de um vetor de endereços e um vetor de dados ou locais de memória, onde cada local está associado a um endereço. O modelo implementado pelo SDM-Go utiliza os vetores de bits que representam estados do tabuleiro como endereços. Os valores associados a cada vetor de bits, ou protótipo, são efetivamente os dados armazenados nos locais de memória. A Figura 4.22 ilustra o modelo implementado no SDM-Go.

O modelo da memória SDM implementado também possui um registrador de endereços, um registrador de entrada e um registrador de saída. Além destes, há ainda o vetor de seleção, que indica quais locais de memória estão selecionados. O registrador

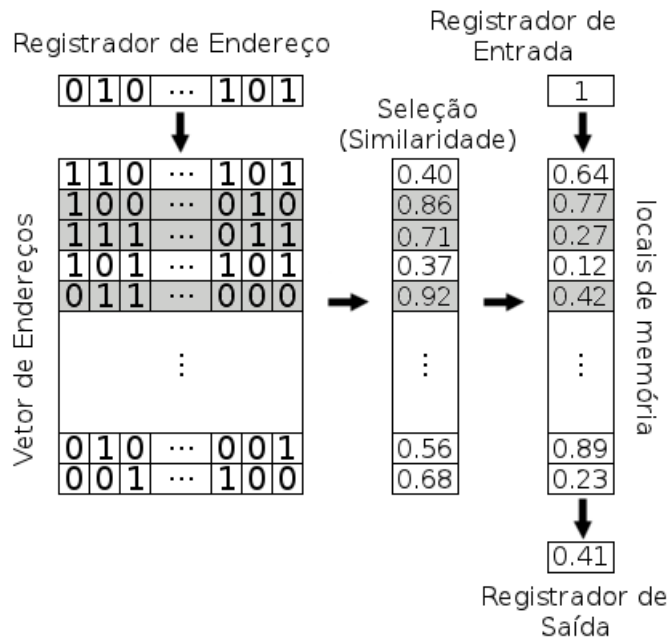


Figura 4.22: Modelo da memória SDM implementada pelo agente SDM-Go.

de endereços é usado para armazenar um dado vetor de bits *VB1* que será comparado com os demais protótipos armazenados na memória. A comparação é feita de acordo com o pseudocódigo do algoritmo apresentado na Figura 4.15 na Seção 4.1.3.1. O grau de similaridade entre o vetor *VB1* e cada protótipo é armazenado no vetor de seleção. No modelo apresentado na Figura 4.22, apenas os endereços e locais de memória cujo grau de similaridade é maior que 0.7 (70%) são selecionados. Os locais de memória selecionados são submetidos a uma operação de entrada ou saída. No caso de uma operação de entrada, os valores dos locais de memória são atualizados a partir do valor armazenado no registrador de entrada. Por outro lado, no caso de uma operação de saída, o valor resultante, armazenado no registrador de saída é obtido a partir do processamento dos valores dos locais de memória. A maneira exata como as operações de entrada e saída acontecem é explicada mais à frente, nas seções 4.2 4.3.

O modelo de memória ilustrado na Figura 4.22 é implementado pelo SDM-Go na linguagem *C++* através da seguinte estrutura de dados:

```

1 struct SDMEntry
2 {
3     struct BitBoardRepresentation address;
4     double similarity;
5     struct SDMLocation mem_location;
6     unsigned int hits;
7 };

```

A estrutura *SDMEntry* corresponde a uma entrada do modelo de memória apresentado anteriormente. Cada membro da estrutura corresponde a um elemento do modelo ilustrado na Figura 4.22, a exceção da variável *hits*, conforme explicado a seguir:

- **address**, esta variável do tipo *BitBoardRepresentation*, cujos detalhes são explicados antes na Seção 4.1.2.1, é uma representação em bits de um estado de tabuleiro armazenado ou um protótipo, que corresponde a uma entrada do vetor de endereços;
- **similarity**, este campo da estrutura corresponde a uma entrada do vetor de seleção, sendo utilizado para indicar o grau de similaridade entre o protótipo e um dada representação de bits. O valor contido na variável é um número entre 0 e 1 que indica a porcentagem de semelhança entre dois vetores de bits;
- **mem_location**, os dados referentes ao protótipo, que faz o papel de endereço deste local de memória, são armazenados nesta variável. Como mais informações além do valor associado ao protótipo são armazenadas, utiliza-se a estrutura *SDMLocation* que é explicada em maiores detalhes a seguir no texto;
- **hits**, esta variável não corresponde a nenhum elemento do modelo de memória apresentado. Ela é utilizada para contar quantas vezes um dado protótipo é selecionado durante a busca na fase de construção da árvore.

Note que os registradores de endereço, entrada e saída não são contemplados com uma variável específica na implementação do SDM-Go. Na realidade, os registradores correspondem simplesmente a parâmetros passados como argumentos para os procedimentos que acessam a memória SDM. No modelo de memória apresentado na Figura 4.22 os locais de memória correspondem apenas ao valor associado a um dado protótipo, entretanto, na implementação do SDM-Go outra informação também está presente. O SDM-Go necessita armazenar além do valor associado a um protótipo, o número de vezes que este valor foi atualizado. Desta maneira, os locais de memória são implementados através da estrutura *SDMLocation*, apresentada a seguir:

```
1 struct SDMLocation
2 {
3     double value;
4     double count;
5 };
```

A estrutura *SDMLocation* contém apenas dois campos:

- **value**, este campo armazena o valor associado ao protótipo que é utilizado para calcular a avaliação prévia de estados de tabuleiro similares durante a fase de construção da árvore da busca;
- **count**, é uma variável que funciona como contador. Entretanto, diferente do campo *hits*, pertencente a estrutura *SDMEntry* e explicado anteriormente, esta variável conta quantas vezes um certo protótipo é selecionado na fase de retropropagação da busca. Quando isto ocorre, uma operação de escrita atualiza o valor associado ao protótipo.

A memória SDM é formada, então, de um vetor de estruturas *SDMEntry* que implementa uma entrada individual da memória. Assim, através da seguinte declaração na classe que efetivamente implementa a memória SDM e suas operações o modelo apresentado na Figura 4.22 é criado:

```

1 /** Código inicial da classe [...] */
2
3 std::vector<SDMEntry> sdm;
4
5 /** [...] código restante da classe */

```

Como não é necessário ordenar as entradas da memória SDM, a estrutura de dados *vector* é utilizada. Presente na biblioteca padrão da linguagem *C++*, daí a inclusão do *namespace std* antes, o tipo *vector* implementa um contêiner sequencial de tipos abstratos *T* (*vector<T>*). Os elementos são armazenados em locais de memória contíguos e podem ser acessados através de ponteiros e *offsets*, da mesma maneira que um *array*. Entretanto, o tamanho de um *vector* varia dinamicamente, o que aumenta o consumo de memória em relação a um *array*. Comparado aos outros contêineres sequenciais oferecidos pela biblioteca padrão da linguagem *C++*, o *vector* é bastante eficiente quando se trata de acessar elementos e adicioná-los no final da sequência. Outra vantagem é a possibilidade de aumentar dinamicamente o tamanho da memória esparsamente distribuída. No presente trabalho, a memória pode aumentar até um limite máximo estabelecido em 100000 protótipos. É importante deixar claro que o espaço requerido para armazenar o tamanho máximo é bastante reduzido devido à representação em bits utilizada. No caso do maior tabuleiro, 19X19, são necessários 91 bytes para cada protótipo, conforme apresentado anteriormente pela tabela 4.3. Assim, 100000 protótipos ocupam 9100000 bytes ou, aproximadamente, 8.6 Megabytes de memória. Não há necessidade de ordenar os elementos, pois ao executar os testes de similaridade, todos os elementos, protótipos da memória, necessitam ser comparados. O SDM-Go utiliza o grau de similaridade para selecionar alguns protótipos da memória cujos valores serão utilizados com dois propósitos, dependendo da fase da busca:

- Na fase de construção da árvore, os valores dos protótipos selecionados são utilizados para computar a avaliação prévia do nó que o vetor de bits representa;
- Na fase de retropropagação, o resultado da simulação também é utilizado para atualizar os valores dos protótipos, além dos valores dos nós do caminho percorrido.

O processo para selecionar os protótipos que mais se assemelham com um dado vetor de bits, que representa um nó da árvore, é explicado a seguir na Seção 4.1.4.1.

4.1.4.1 Seleção de Protótipos da Memória

Durante as fases de construção da árvore e retropropagação da busca, o SDM-Go utiliza a memória SDM para avaliar novos nós, no primeiro caso e para atualizar os valores dos protótipos, no segundo caso. Em ambas as situações, os protótipos e seus respectivos valores envolvidos no processo são escolhidos de acordo com o grau de similaridade com os nós da árvore. O procedimento para calcular o grau de similaridade utiliza a distância de Hamming computada por operações *bitwise*, conforme apresentado anteriormente nas seções 4.1.3.1 e 4.1.3.2. Utilizando o grau de similaridade calculado conforme o pseudocódigo da Figura 4.15 da Seção 4.1.3.1, as entradas da memória SDM que atendem a um critério mínimo são selecionadas. Este processo é apresentado no pseudocódigo da Figura 4.23 a seguir.

SelecionaPrototipos(*VetorBits*) :

```

1: ProtSimilares = {}
2: TamanhoSDM = Tamanho(SDM)
3: for all indice ∈ [0..TamanhoSDM] do
4:   Prototipo = SDM[indice]
5:   Similaridade = CalculaSimilaridade(Prototipo, VetorBits)
6:   if Similaridade ≥ 0.7 then
7:     EscreveSimilaridade(SDM, Prototipo, Similaridade)
8:     Adiciona(indice, ProtSimilares)
9:   end if
10: end for
11: return ProtSimilares

```

Figura 4.23: Pseudocódigo do Procedimento que Seleciona os Protótipos Mais Similares.

O procedimento *SelecionaPrototipos* retorna uma lista com os índices dos protótipos selecionados. Posteriormente, estes índices são utilizados para acessar diretamente as entradas da memória SDM. Cada linha do procedimento é explicada a seguir:

Linha 1. A lista que armazena os índices dos protótipos selecionados é iniciada com a lista vazia.

Linha 2. O tamanho da SDM é obtido para limitar o laço que se inicia na linha seguinte.

Linha 3. O vetor de entradas da SDM é percorrido da primeira até a última posição.

Linha 4. Cada protótipo da SDM é obtido a partir do índice de sua respectiva entrada na memória.

Linha 5. O grau de similaridade entre o protótipo e o vetor de bits é calculada.

Linha 6. O teste verifica se o grau de similaridade atende ao critério mínimo pré-estabelecido de 0.7 (70%).

Linha 7. O grau de similaridade é escrito no campo da entrada correspondente ao protótipo, que por atender ao critério mínimo de 70% é então selecionado.

Linha 8. O índice da entrada da memória cujo protótipo selecionado pertence é ar-

mazenado na lista.

Linha 11. Após percorrer todas as entradas da memória SDM, os índices dos protótipos selecionados, armazenados na lista *ProtSimilares*, são retornados.

Assim, durante as fases de construção da árvore e retropropagação, o SDM-Go é capaz de obter quais protótipos mais se assemelham a um determinado vetor de bits que representa um nó da árvore. Em ambas as fases da busca, os valores dos protótipos são utilizados com propósitos diferentes. As duas seções a seguir, 4.2 e 4.3, detalham a maneira como os valores são utilizados e com quais objetivos, durante as fases de construção da árvore e retropropagação, respectivamente.

4.2 Utilização da Memória Esparsamente Distribuída na Fase de Construção da Árvore da Busca

Durante a fase de construção da árvore, a busca MCTS executada pelo SDM-Go percorre um caminho na árvore selecionando os nós de acordo com sua avaliação. A seleção continua até o momento quando um nó s cujos filhos não estão ainda inseridos na árvore é encontrado. Neste momento, inicia-se a expansão, quando os nós s_i filhos de s são gerados e adicionados na árvore. Como os nós s_i nunca foram visitados durante algum episódio da busca, e portanto não possuem avaliação, não é possível distinguir os nós que representam ações boas daqueles que representam ações ruins. Desta maneira, em um primeiro momento, qualquer nó será escolhido para dar início a fase de *playout* sem qualquer indicação de benefício para a busca da melhor ação. Para contornar esta situação e melhorar a decisão da busca ao selecionar um nó gerado na fase de construção da árvore, o Fuego utiliza uma abordagem denominada *Prior Knowledge*, ou **Avaliação Prévia**, em português. De maneira sucinta, o agente utiliza várias heurísticas que se resumem a verificar a presença de vários padrões de peças nos tabuleiros correspondentes aos nós s_i gerados na expansão. Estes padrões de peças são parte do conhecimento humano, alcançado após muitos anos de estudo e prática do jogo de Go. São configurações sabidamente importantes que representam vantagem para um determinado jogador e, conseqüentemente, desvantagem para seu oponente. Portanto, ao verificar a presença de uma destas configurações de peças em algum tabuleiro representado por algum nó s_i , é possível atribuir uma certa avaliação ao nó, dependendo da vantagem ou desvantagem que a heurística representa do ponto de vista do jogador. Esta abordagem, como pode ser percebido, é fortemente supervisionada e atrelada ao domínio, pois depende da intervenção humana direta na codificação de padrões de peças específicos do jogo de Go para formar as heurísticas do *Prior Knowledge*. De maneira distinta, o agente SDM-Go reduz a supervisão do Fuego ao substituir as heurísticas de *Prior Knowledge* por uma estratégia inteligente não supervisionada e independente de domínio. Na abordagem do SDM-Go, a **Avaliação Prévia** dos nós s_i

é obtida a partir dos valores de configurações de tabuleiro já visitadas durante a busca e armazenados na memória SDM. Cada novo nó s_i , gerado na expansão, é comparado com cada protótipo P_j da memória, obtendo-se o grau de similaridade entre ambos. Sempre que o grau de similaridade entre um protótipo P_j e um nó s_i atende a um critério mínimo, neste trabalho estabelecido em 70%, o valor de P_j é utilizado no cálculo da avaliação prévia de s_i . Caso um nó s_i não possua um grau de similaridade mínimo, 70% ou mais, com algum protótipo armazenado na memória SDM, nenhum valor é utilizado e a avaliação prévia de s_i é 0. Neste caso, o vetor de bits correspondente a s_i é armazenado na memória SDM e se torna um novo protótipo, de maneira a suprir a falta de protótipos similares a esta configuração de tabuleiro. A maneira como os valores dos protótipos são utilizados para computar a avaliação prévia dos nós gerados na fase de construção da árvore é explicada em maiores detalhes a seguir na Seção 4.2.1.

4.2.1 Cálculo da Avaliação Prévia do Nó

Como explicado anteriormente, a avaliação prévia de novos nós gerados durante a fase de construção da árvore é obtida a partir dos valores dos protótipos armazenados na memória SDM. A seleção do conjunto de protótipos P_j cujos valores serão utilizados no cálculo da avaliação do novo nó s_i é feita de acordo com o grau de similaridade. Os protótipos que atendem ao critério mínimo de 70% de similaridade com o nó s_i são selecionados e seus respectivos valores são utilizados no cálculo da avaliação de s_i . A equação 4.1 define este cálculo:

$$Q(s, a) = \frac{(\sum_{j=1}^J A_j * \mu(j, s_i))}{J} \quad (4.1)$$

Onde, $Q(s, a)$ é a avaliação de s_i , gerado a partir da execução da ação a em s , J é o número total de protótipos selecionados, A_j é o valor associado ao protótipo P_j e $\mu(j, s_i)$ é o grau de similaridade entre o protótipo P_j e s_i . Assim, além de funcionar como critério de seleção, o grau de similaridade também é utilizado para ponderar o impacto que o valor A_j tem no cálculo da equação 4.1. Logo, os valores dos protótipos mais similares tem maior impacto no valor final da avaliação prévia de s_i . O procedimento utilizado pelo SDM-Go para calcular a avaliação prévia de um nó s_i , representado por seu respectivo vetor de bits, é apresentado no pseudocódigo da Figura 4.24 a seguir.

O procedimento *AvaliaNo* recebe como parâmetro o vetor de bits que corresponde ao nó a ser avaliado. Cada linha é explicada a seguir:

Linha 1. O procedimento *SelecionaPrototipos*, explicado anteriormente na Seção 4.1.4.1, retorna uma lista com os índices dos protótipos que atendem ao grau mínimo de similaridade com o vetor passado como parâmetro.

Linha 2. A variável *NroPrototipos* é utilizada como contador do número de protótipos selecionados e cujos valores são usados no cálculo da avaliação. Aqui a variável é iniciada

AvaliaNo(*VetorBits*) :

```

1: ProtSimilares = SelecionePrototipos(VetorBits)
2: NroPrototipos = 0
3: Avaliacao = 0
4: Somatorio = 0
5: for all indice ∈ ProtSimilares do
6:   EntradaSDM = SDM[indice]
7:   Similaridade = ObtemSimilaridade(EntradaSDM)
8:   Valor = ObtemValor(EntradaSDM)
9:   Somatorio = Somatorio + (Valor * Similaridade)
10:  NroPrototipos = NroPrototipos + 1
11: end for
12: if NroPrototipos > 0 then
13:   Avaliacao = Somatorio/K
14: else AdicionaPrototipo(SDM, VetorBits)
15: end if
16: return Avaliacao

```

Figura 4.24: Pseudocódigo do procedimento utilizado pelo SDM-Go para calcular a avaliação prévia de um nó.

com o valor 0.

Linha 3. A variável de retorno da avaliação do nó é iniciada também como o valor 0.

Linha 4. A variável auxiliar que acumula o somatório envolvido no cálculo da avaliação, conforme mostrado na equação 4.1, também é iniciada com 0.

Linha 5. Todos os índices dos protótipos selecionados são iterados no laço que inicia nesta linha.

Linha 6. A entrada da SDM correspondente ao protótipo selecionado é obtida a partir de um índice da lista *ProtSimilares*.

Linha 7. O grau de similaridade armazenado na entrada da SDM é obtido a partir da chamada nesta linha.

Linha 8. A entrada da memória, selecionada a partir do índice, fornece também o valor associado ao protótipo similar ao vetor de bits.

Linha 9. O valor associado ao protótipo, ponderado pela similaridade, é adicionado ao valor do somatório acumulado até o momento.

Linha 10. O número de protótipos cujos valores são utilizados no cálculo da avaliação é incrementado por 1, ao final da iteração.

Linha 12. O teste verifica se algum protótipo foi selecionado e, conseqüentemente, seu valor utilizado para calcular a avaliação do nó.

Linha 13. No caso da condição do teste anterior ser verdadeira, o valor final obtido pela equação 4.1 é computado através da divisão do somatório pelo número de protótipos selecionados. Este valor representa a avaliação prévia do nó representado pelo vetor de bits passado como parâmetro.

Linha 14. Caso o teste da **linha 12** falhe, isto significa que nenhum protótipo atendeu ao grau mínimo de similaridade com o vetor de bits. Portanto, este vetor se torna um novo protótipo e é inserido na memória SDM.

Linha 16 O valor da avaliação prévia do nó é retornado ao final do procedimento.

O pseudocódigo apresentado na Figura 4.24 implementa a equação 4.1 e calcula a avaliação prévia de um dado nó, representado pelo seu respectivo vetor de bits. É importante ressaltar que os valores armazenados na memória SDM são sempre sob a ótica do jogador das peças pretas. Isto acontece para manter a consistência dos valores armazenados que desta maneira representam sempre a avaliação a partir de um único ponto de vista. Além disso, é simples lidar com as situações onde a cor do movimento buscado é branca. Sempre que um nó é avaliado sob a perspectiva das peças brancas, o valor obtido necessita ser invertido. Esta inversão acontece de maneira simples, subtraindo de 1 a avaliação retornada pelo procedimento, segundo a equação 4.2 a seguir:

$$Q(s, a)^{-1} = 1 - Q(s, a) \quad (4.2)$$

Assim, o valor 1 que corresponde à certeza de vitória das peças pretas, ao ser invertido pela equação 4.2 se torna o valor 0, devidamente correspondendo à certeza de derrota das peças brancas. Por outro lado, se o valor calculado é 0, derrota certa para as peças pretas, a inversão o transforma no valor 1, significando certeza de vitória para as peças brancas. Os demais valores possíveis, compreendidos no intervalo entre 0 e 1, indicam a probabilidade de vitória das peças pretas. Ao serem invertidos pela equação 4.2, estes passam a indicar a probabilidade de vitória das peças brancas. É interessante ainda ressaltar que, caso nenhum protótipo atenda ao grau mínimo de similaridade com o vetor de bits, o valor da avaliação prévia do nó, retornado pelo procedimento *AvaliaNo* é 0, pois não existe nenhum protótipo cujo valor possa ser utilizado. Neste caso, o vetor de bits que representa este nó é imediatamente inserido na memória SDM como um novo protótipo. Novos protótipos são sempre inseridos na memória durante a fase de construção da árvore. Da mesma forma que não é possível avaliar o nó, também não existem informações disponíveis para atribuir alguma avaliação ao novo protótipo e inicialmente seu valor é 0. O fato de que o valor do termo de qualidade seja 0 não tem impacto na avaliação UCT pois a frequência de visitas do nó também é 0 e assim ele tem preferência sobre nós ainda não visitados. Posteriormente, durante a fase de retropropagação, o valor do protótipo será atualizado, conforme explicado a seguir na Seção 4.3.

4.3 Estendendo os Resultados das Simulações durante a Fase de Retropropagação

Conforme explicado anteriormente na Seção 4.2, na fase de construção da árvore, os nós que são gerados e adicionados à árvore têm sua avaliação prévia calculada com base nos valores dos protótipos armazenados na memória SDM. Após este cálculo, um nó é escolhido e a partir dele inicia-se a fase de *playout*, quando uma política $\pi_{\text{playout self-play}}$ escolhe ações de forma a conduzir o jogo rapidamente até um estado final s_f . O estado s_f é então avaliado e obtém-se o resultado R da simulação (vitória, empate ou derrota). Este resultado é utilizado na fase de retropropagação para atualizar os valores dos nós percorridos durante a fase de construção da árvore do episódio corrente. Além deste processo que ocorre durante a retropropagação, o SDM-Go estende a utilização do resultado R da simulação para atualizar também os valores dos protótipos armazenados na memória esparsamente distribuída. Assim como na fase de construção da árvore, o grau de similaridade é utilizado como critério para selecionar os protótipos cujos valores receberão a atualização. Para cada nó s do caminho percorrido na árvore, cujo valor será atualizado, o SDM-Go calcula o grau de similaridade com cada protótipo P_j armazenado na memória SDM. Novamente, são selecionados aqueles protótipos que obedecem ao critério de possuir um grau de similaridade de pelo menos 70%. Conforme descrito anteriormente na Seção 4.2, durante a fase de construção da árvore, novos nós são gerados pela expansão e pode ocorrer que nenhum dos protótipos armazenados na memória atenda ao critério mínimo de similaridade com algum novo nó s_k . Neste caso, o vetor de bits correspondente ao nó s_k se torna um novo protótipo P_k que é adicionado à memória SDM e seu valor associado é 0, pois, não existem ainda informações para avaliar a qualidade do estado de tabuleiro que ele representa. Após a inclusão do novo protótipo P_k e o final da expansão, a fase de *playout* inicia-se a partir de algum nó s_i . Caso este nó não seja s_k , cuja representação em bits se tornou o novo protótipo P_k , s_i é certamente um irmão de s_k e faz parte do caminho percorrido durante a fase de construção da árvore. Portanto, o valor de s_i é atualizado a partir do resultado final da simulação durante a retropropagação. É possível, então, saber com certeza que dentre estes protótipos selecionados está o novo protótipo P_k adicionado durante a construção da árvore. Isto acontece porque sendo o nó s_i , pertencente ao caminho percorrido na árvore, irmão de s_k , o grau de similaridade entre eles aproxima-se de 100%. Caso s_i seja o próprio nó s_k , o grau de similaridade é de fato 100%. Desta maneira, apesar de que sempre que um protótipo é inserido na memória SDM seu valor correspondente é 0, garantidamente ele será atualizado em seguida, durante a retropropagação. O cálculo das atualizações dos valores dos protótipos a partir do resultado da simulação é explicado a seguir na Seção 4.3.1.

4.3.1 Cálculo da Atualização dos Valores dos Protótipos

Durante a fase de retropropagação, o resultado da simulação também é utilizado para atualizar os valores dos protótipos P_j que atendem ao critério mínimo de similaridade com algum nó s pertencente ao caminho percorrido na árvore. O valor A_j associado ao protótipo P_j é atualizado a partir do resultado R da simulação de acordo com a equação 4.3 a seguir:

$$A'_j = \frac{(R * \mu(j, s) + A_j)}{(N + 1)} \quad (4.3)$$

Onde $\mu(j, s)$ é o grau de similaridade entre o protótipo P_j e o nó s e N é o número de vezes que o protótipo foi atualizado anteriormente. A'_j é o novo valor de P_j calculado conforme a equação 4.3. De maneira similar ao cálculo efetuado na equação 4.1 na fase de construção da árvore, além de ser utilizado como critério para a seleção dos protótipos, o grau de similaridade também pondera o impacto do resultado R da simulação. Quanto maior a similaridade entre o protótipo P_j e o nó s , maior é a atualização do valor de P_j a partir de R .

Cabe ainda lembrar que, conforme explicado anteriormente na Seção 4.2.1, os valores dos protótipos são sempre armazenados na memória sob a ótica do jogador das peças pretas. Logo, caso a busca seja executada para encontrar o melhor movimento para as peças brancas, então o resultado R da simulação é invertido de acordo com a fórmula 4.4 a seguir:

$$R^{-1} = 1 - R \quad (4.4)$$

Assim, se a busca é executada para encontrar o melhor movimento para as peças brancas, o resultado R com valor 1 indica vitória para as peças brancas. Ao passar pela inversão da equação 4.4, R , cujo valor agora é 0, reflete o resultado a partir da ótica das peças pretas, significando derrota. E ainda na ótica das peças brancas, se R possui o valor 0, o que significa derrota, a inversão da equação 4.4, transforma o valor de R para 1, adequadamente refletindo vitória das peças pretas. Os demais valores que R pode assumir, compreendidos entre 0 e 1, indicam a probabilidade de vitória. A inversão deste valor pela equação 4.4 produz a probabilidade de vitória da cor oposta. Desta maneira, sempre que o resultado R da simulação se refere às peças brancas, ele é invertido para atualizar os protótipos de acordo com a ótica das peças pretas. O pseudocódigo do procedimento implementado pelo SDM-Go para atualizar os valores dos protótipos durante a fase de retropropagação é apresentado a seguir na Figura 4.25.

Cada linha do pseudocódigo do procedimento *AtualizaPrototipos* é explicada em maiores detalhes agora:

Linha 1. A chamada do procedimento *SelecionaPrototipos* retorna uma lista com os

AtualizaPrototipos(*VetorBits*, *Resultado*) :

```

1: ProtSimilares = SelecionaPrototipos(VetorBits)
2: for all indice ∈ ProtSimilares do
3:   EntradaSDM = SDM[indice]
4:   Similaridade = ObtemSimilaridade(EntradaSDM)
5:   Valor = ObtemValor(EntradaSDM)
6:   NroAtualizacoes = ObtemNroAtualizacoes(EntradaSDM)
7:   NovoValor = (Valor + Resultado * Similaridade) / NroAtualizacoes + 1
8:   EscreveValor(EntradaSDM, NovoValor)
9:   EscreveNroAtualizacoes(EntradaSDM, NroAtualizacoes + 1)
10: end for
```

Figura 4.25: Pseudocódigo do procedimento para atualizar os valores dos protótipos armazenados na memória SDM.

índices dos protótipos similares ao nó cujo valor é atualizado durante a retropropagação.

Linha 2. O laço que se inicia nesta linha itera sobre cada índice da lista de protótipos.

Linha 3. A partir do índice do protótipo, obtém-se a entrada na memória SDM.

Linha 4. O grau de similaridade entre o protótipo e o nó pertencente ao caminho da árvore é obtido da entrada da memória.

Linha 5. O valor associado ao protótipo é obtido a partir da entrada da memória.

Linha 6. O número de vezes que o valor do protótipo foi atualizado está armazenado na memória. Ele é obtido para ser utilizado no cálculo da atualização.

Linha 7. O novo valor do protótipo é calculado conforme a equação 4.3, apresentada antes.

Linha 8. Após computar a atualização do valor do protótipo, este é escrito na memória.

Linha 9. O número de vezes que o valor do protótipo foi atualizado é incrementado em 1 e armazenado na respectiva entrada da memória.

O procedimento *AtualizaPrototipos* é executado para cada nó cujo valor é atualizado durante a fase de retropropagação. O resultado *R* da simulação, utilizado para atualizar os valores dos protótipos, representa sempre a visão do jogador das peças pretas. Desta maneira, mantém-se a consistência dos valores armazenados. Conforme explicado anteriormente, é fácil inverter estes valores, caso a busca seja feita para escolher o melhor movimento para as peças brancas.

4.4 Limitações da Utilização da Memória SDM para Avaliar os Nós

Apesar de o uso da memória SDM ser uma alternativa viável ao processo fortemente supervisionado do agente Fuego, é importante destacar que esta abordagem possui algumas

limitações. O critério de similaridade baseado na distância de Hamming, em alguns casos, pode não ser o mais apropriado para avaliar os novos nós gerados na fase de construção da árvore. Existem situações locais complexas cujo impacto no resto do tabuleiro se estende por longos períodos durante o jogo. Por exemplo, a Figura 4.26 a seguir apresenta uma destas situações.

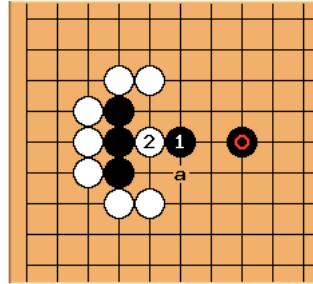


Figura 4.26: Exemplo do movimento *tesuji* *Crane's Nest*.

No tabuleiro apresentado na Figura, após a jogada da peça preta marcada **1**, a resposta apropriada por parte das peças brancas é o movimento cuja peça é marcada por **2**. Este movimento é chamado de *tesuji*, pois nesta configuração de peças é o único movimento que impede que as três peças pretas cercadas sejam salvas da captura. Qualquer outro movimento diferente das peças brancas permite que estas peças pretas sejam conectadas com o exterior e portanto sejam salvas. Durante uma partida de Go é comum ocorrer diversas situações onde é necessário descobrir o movimento *tesuji* que garante um resultado local incontestável, como, por exemplo, no caso da Figura 4.26, que as peças pretas sejam capturadas. Entretanto, a representação global do tabuleiro nos vetores de bits não é apropriada para capturar a característica decisiva deste tipo de situação. Na configuração de peças do tabuleiro mostrada na Figura 4.26 o vetor de bits VB_2 que representa o tabuleiro após o movimento *tesuji* **2** é bastante similar ao vetor VB_a que representa outro movimento, como por exemplo colocar uma peça branca no ponto marcado *a*. De fato, considerando que o restante do tabuleiro é o mesmo, a diferença entre os dois vetores seria de apenas 1 bit. Desta maneira, ao avaliar ambos vetores VB_2 e VB_a , os valores atribuídos a eles serão bastante similares. Entretanto, eles representam situações qualitativamente muito diferentes e em uma análise mais cuidadosa o movimento **2** deveria receber uma avaliação muito melhor que *a*. No capítulo 6 algumas melhorias para contornar esta dificuldade do SDM-Go para avaliar tais situações são propostas.

Capítulo 5

Experimentos e Resultados

Para verificar o ganho obtido com a aplicação da memória SDM no processo de busca pelo agente SDM-Go, foram realizados três cenários de testes, conforme descrito a seguir. Os cenários foram escolhidos de maneira a estimar a melhoria que a memória SDM introduz na busca em árvore MC. Em todos os cenários de testes, o SDM-Go jogou em torneios contra dois oponentes distintos: (a) Fuego sem *prior knowledge* e (b) Fuego com *prior knowledge*, que de agora em diante serão referenciados apenas por *a* e *b*, respectivamente. Em todos os torneios contra o oponente *a*, o SDM-Go também joga sem utilizar a estratégia *Prior Knowledge*. Por outro lado, quando o SDM-Go joga contra o oponente *b*, ele utiliza *Prior Knowledge* até o movimento *M*, quando esta estratégia é substituída no SDM-Go pelo uso da memória SDM. O valor do movimento *M* varia de acordo com o cenário de teste. Durante os torneios, o SDM-Go foi testado jogando tanto com as peças pretas quanto com as peças brancas, alternando as cores a cada novo jogo. Desta maneira, o agente joga um número igual de jogos com cada uma das cores. O grau de similaridade mínimo exigido para utilizar os valores da memória SDM é de 70%. A constante *C* da equação 2.4 foi ajustada com o valor de 0.7, que é o padrão utilizado pelo agente Fuego [Müller 2013]. Em todos os cenários, o número de vitórias do SDM-Go sobre seus oponentes é a medida utilizada para avaliar o desempenho obtido com a utilização da SDM. O *hardware* utilizada para executar os cenários de testes consiste em um processador *Intel Core 2 Quad* 2.4 GHz com 4GB de memória RAM.

No primeiro cenário, o SDM-Go jogou contra os oponentes *a* e *b* em torneios de 1000 jogos, para o tabuleiro 9x9, 500 jogos, para o tabuleiro 13x13 e 200 jogos para o tabuleiro 19x19. Neste cenário, a memória SDM é utilizada a partir do 60º movimento. O número de simulações efetuadas pela busca MC foi limitado a 500 para todos os agentes. Esta limitação visa estabelecer uma igualdade de recursos para fins de comparação dos agentes, bem como permitir um tempo razoável de execução dos testes em face dos recursos computacionais disponíveis. A Tabela 5.1 mostra os resultados obtidos pelo SDM-Go contra cada oponente. A taxa de vitórias de 53.9% do SDM-Go contra o oponente *a* (Fuego sem *prior knowledge*) demonstra que a utilização da SDM tem um impacto positivo no

processo de busca MC do SDM-Go, já que esta é a única diferença nos recursos utilizados por ambos agentes. No segundo torneio apresentado na Tabela 5.1, o SDM-Go joga contra o oponente b , que utiliza *Prior Knowledge*. Para manter um critério de igualdade na comparação, o SDM-Go também utiliza este recurso. Porém, diferentemente de seu oponente, este é apenas utilizado até o 60º movimento, quando é substituído pelo uso da SDM. Neste caso, verifica-se pelos resultados, que o SDM-Go, apesar de não vencer a maioria dos jogos, obtendo uma taxa de vitórias de 47.4%, ainda é suficientemente competitivo. A utilização da SDM cumpre seu objetivo de compensar a inserção da supervisão caracterizada pela técnica *Prior Knowledge* do agente Fuego. Os resultados da Tabela 5.1 mostram também que nos torneios envolvendo tabuleiros 9x9, a média de tempo de execução do SDM-Go é 110% superior do que a de seus oponentes (0.42 segundos contra 0.2 segundos). Isto se dá obviamente pelo custo de utilizar a memória SDM, que exige que muitos testes de similaridade seja executados para determinar o grau de similaridade entre os nós da árvore de busca e os protótipos armazenados. Os resultados dos torneios da Tabela 5.1 envolvendo tabuleiros 13x13 mostram que o SDM-Go mantém um nível bastante competitivo, obtendo aproximadamente 49% de vitórias contra o oponente a . Entretanto, isto não ocorre contra o oponente b quando a taxa de vitórias é reduzida para aproximadamente 27%. O mesmo comportamento é observado nos torneios envolvendo tabuleiros 19x19, onde apesar de que o SDM-Go apresenta apenas uma pequena redução na taxa de vitórias contra o oponente a (aproximadamente 44% de vitórias), quando joga contra o oponente b , ele se torna bastante inferior com uma taxa de vitórias de aproximadamente 12%. A partir destes resultados contra o oponente b fica claro que, em tabuleiros maiores, a estratégia *Prior Knowledge* empregada pelo jogador Fuego se sobressai muito mais que a memória SDM utilizada pelo SDM-Go. Nestes tabuleiros maiores, é possível que a representação em bits usada pela memória SDM não dê a devida importância às situações locais que possuem um impacto a longo prazo no jogo, como explicado na Seção 4.4. A medida que o tamanho do tabuleiro fica maior, diminui a capacidade da representação em bits de capturar a característica decisiva de problemas locais como, por exemplo, os movimentos *tesuji*. Para lidar com este tipo de problema, algumas soluções são propostas no capítulo 6. Por outro lado, considerando os resultados dos torneios contra o oponente a , verifica-se que a grande desvantagem do SDM-Go é em relação ao tempo de execução da busca (556% no tabuleiro 13x13 e 980% no tabuleiro 19x19). Conforme mencionado anteriormente, este aumento no tempo de processamento é devido aos vários testes de similaridades necessários para o funcionamento da memória SDM. Contudo, este processamento maior é o que possibilita o sucesso do SDM-Go em obter um processo de aprendizado independente de domínio. Considerando o baixo desempenho do SDM-Go em tabuleiros maiores, o segundo cenário de testes investiga a utilização da memória SDM a partir de movimentos posteriores ao 60º movimento usado no primeiro cenário. Neste segundo cenário, a memória SDM é empregada a partir do 160º movimento para o tabu-

leiro 13x13 e a partir do 240º movimento para o tabuleiro 19x19. Assim como no primeiro cenário, são utilizados 500 jogos nos torneios no tabuleiro 13x13 e 200 jogos naqueles no tabuleiro 19x19. Os resultados apresentados na Tabela 5.2 mostram uma melhora significativa do SDM-Go, principalmente no que diz respeito ao oponente *b*. Quando o agente joga no tabuleiro 13x13, a taxa de vitórias se iguala a do oponente *a* e contra o oponente *b* o nível de competitividade do SDM-Go aumenta consideravelmente em relação ao cenário anterior, mantendo-se bastante próximo do desempenho do oponente (47.4%). Para o tabuleiro 19x19, apesar de que a melhora não é tão expressiva quanto no tabuleiro 13x13, ainda assim o desempenho do agente melhora, chegando a praticamente dobrar sua taxa de vitórias contra o oponente *b* em relação ao cenário anterior (21.4%). A melhora no desempenho geral do agente SDM-Go nestes tabuleiros maiores, sem dúvida se deve à utilização da memória SDM a partir de movimentos mais avançados no decorrer do jogo. Nesta situação, a memória SDM está menos sujeita aos problemas relacionados com a dificuldade da representação em bits de capturar a importância de situações locais que têm grande impacto na situação global do jogo.

Como pode ser percebido pelos resultados das Tabelas 5.1 e 5.2, o mérito do agente Fuego reside na utilização da estratégia *Prior Knowledge*. Nos torneios em que esta estratégia não foi usada, o SDM-Go alcança taxas de vitórias superiores ou bastante competitivas. De fato, uma grande dificuldade do jogo de Go, experimentada tanto por agentes automáticos como por jogadores humanos, é a complexidade das jogadas e seus resultados que muitas vezes são visíveis apenas a longo prazo. Há sempre a necessidade de estudar cuidadosamente os padrões de jogadas e as formas que estes criam, de maneira a garantir resultados sólidos e confiáveis em situações locais. Esta abordagem é adotada por todos os jogadores humanos que desejam progredir no entendimento do jogo. Os profissionais e mestres humanos conhecem diversas variações de jogadas em diversas situações, o que possibilita que durante o jogo as muitas opções de movimentação sejam reduzidas, facilitando o *lookahead*. Sem dúvida, este é o objetivo principal da estratégia *Prior Knowledge*, que incorpora o conhecimento humano de formas de peças no tabuleiro em heurísticas usadas para avaliar os novos nós inseridos através da construção da árvore. O SDM-Go, diferentemente, investiga a viabilidade de aprender tal conhecimento de uma maneira não supervisionada, através do histórico de jogos anteriores.

O cenário de testes III tem como objetivo avaliar até que ponto a utilização da memória esparsamente distribuída consegue compensar um aumento gradativo no número de simulações da busca executada pelo oponente *a*. Nos cenários I e II, os agentes utilizaram um limite de 500 simulações na busca de cada novo movimento. Entretanto, no cenário III este limite é estabelecido apenas para o agente SDM-Go, enquanto seu adversário utiliza um número maior de simulações, que varia de 600 a 700 simulações. Os resultados, apresentados na Tabela 5.3 mostram que ao permitir que o oponente opere com um número até 30% maior de simulações, o SDM-Go ainda consegue se manter competitivo com uma

taxa de vitórias superior a 40%. Porém, esta competitividade é comprometida e o desempenho do SDM-Go é reduzido para 31.2% de vitórias quando o oponente aumenta ainda mais o número de simulações, utilizando 50% a mais que o SDM-Go. A partir destes resultados, verifica-se que o SDM-Go permanece competitivo mesmo em situações onde o oponente conta com uma taxa de simulações até 30% maior. A redução do desempenho do SDM-Go quando o número de simulações é 50% maior indica a necessidade de tratar a dificuldade de capturar a importância de certas situações locais através da representação em bits.

O último cenário de testes investiga o impacto no desempenho do agente SDM-Go ao alterar o grau de similaridade utilizado pela memória esparsamente distribuída. Neste cenário, ocorrem torneios de 500 jogos contra o oponente *a*. Nos cenários anteriores, o grau de similaridade utilizado é de 70%. Neste cenário, o valor do grau de similaridade é alterado, variando entre 60% e 90%, em torneios distintos. Os resultados apresentados na Tabela 5.4 mostram que existe um comprometimento entre o valor do grau de similaridade e o desempenho do agente em termos de vitórias. Conforme demonstram os resultados dos torneios, o valor mais apropriado se encontra entre 70% e 80%. De fato, os resultados obtidos mostram que o desempenho do agente começa a ser comprometido ou quando o grau de similaridade aumenta para 90% ou quando é diminuído para 60%. No primeiro caso, com um valor de 90% de grau de similaridade, o que estabelece um critério muito rigoroso para permitir o uso dos valores armazenados na memória SDM. Por outro lado, quando o grau de similaridade é diminuído para 60%, a grande flexibilidade induzida por este parâmetro permite um uso inapropriado da memória SDM. Os experimentos executados mostram que o agente SDM-Go tem sucesso na tarefa de reduzir a característica fortemente supervisionado do agente Fuego, tanto em termos de número de simulações como na utilização da técnica *prior knowledge*, mantendo um nível de desempenho competitivo. No capítulo 6, algumas alternativas de trabalhos futuros são propostas para otimizar estes resultados obtidos até agora.

Tabela 5.1: Taxa de Vitórias do SDM-Go no Cenário de Testes I.

Oponente	Tabuleiro	Taxa de Vitórias do SDM-Go	Tempo
Oponente a	9x9	53.9% (± 1.6)	110%
Oponente b	9x9	47.4% (± 1.6)	110%
Oponente a	13x13	48.5% (± 2.2)	556%
Oponente b	13x13	26.8% (± 2.2)	556%
Oponente a	19x19	41.5% (± 3.4)	980%
Oponente b	19x19	11.7% (± 3.4)	980%

Tabela 5.2: Taxa de Vitórias do SDM-Go no Cenário de Testes II.

Oponente	Tabuleiro	Taxa de Vitórias do SDM-Go
Oponente a	13x13	50% (± 2.2)
Oponente b	13x13	47.6% (± 2.2)
Oponente a	19x19	44.5% (± 3.4)
Oponente b	19x19	21.4% (± 3.4)

Tabela 5.3: Taxa de Vitórias do SDM-Go no Cenário de Testes III (tabuleiro 9x9 contra Fuego sem *prior knowledge*).

Oponente a	Taxa de Vitórias do SDM-Go
600 (20% mais) simulações	43.2% (± 1.6)
650 (30% mais) simulações	40.1% (± 1.6)
750 (50% mais) simulações	31.2% (± 1.6)

Tabela 5.4: Taxa de Vitórias do SDM-Go no Cenário de Testes IV (Tabuleiro 9x9 contra Fuego sem *prior knowledge*).

Grau de Similaridade	Taxa de Vitórias do SDM-Go
90%	50.4% (± 2.2)
80%	53.4% (± 2.2)
75%	52.8% (± 2.2)
60%	49.8% (± 2.2)
50%	46.4% (± 2.2)

Capítulo 6

Conclusão e Trabalhos Futuros

Neste trabalho foi apresentado o SDM-Go, um agente para Go que aprimora a busca em árvore Monte-Carlo através da utilização de uma memória esparsamente distribuída. O jogo de Go é o domínio escolhido devido a sua semelhança com vários problemas reais e a grande complexidade que emerge das regras simples do jogo. A falta de uma boa heurística para o jogo de Go dificulta o emprego de técnicas que obtiveram sucesso em xadrez, damas e gamão, fato que demanda o surgimento de novas abordagens para lidar com o problema. O agente SDM-Go é baseado no agente Fuego e melhora o processo de busca deste através da utilização da memória SDM como recurso de aprendizado. No SDM-Go, o módulo da busca MCTS utiliza recursos de dois outros módulos presentes na arquitetura do agente SDM-Go: a memória SDM e a representação de tabuleiro em vetor de bits. A memória esparsamente distribuída é empregada como uma alternativa ao processo fortemente supervisionado utilizado pelo agente Fuego, onde novos nós encontrados na construção da árvore são previamente avaliados através de heurísticas que correspondem a padrões de peças codificados por profissionais humanos. Diferentemente, o SDM-Go implementa uma abordagem não supervisionada e independente de domínio, onde os valores de tabuleiros previamente visitados durante a busca e armazenados na forma de vetores de bits na memória SDM são utilizados para avaliar novos tabuleiros. As contribuições provenientes da implementação do agente SDM-Go são:

- a implementação de uma representação de tabuleiro baseada em vetores de bits, que é rápida e compacta;
- a utilização da memória SDM para avaliar os valores dos nós da busca MCTS através de conhecimento independente de domínio, obtido a partir do histórico de tabuleiros visitados durante a busca;
- a extensão do emprego do resultado da simulação para atualizar, além dos valores dos nós percorridos na árvore de busca, também os valores dos protótipos armazenados na memória SDM.

Os resultados dos experimentos mostram que a memória SDM conseguiu desempenhar com sucesso a tarefa de prover uma abordagem independente de domínio para avaliar previamente o valor dos nós encontrados na construção da árvore de busca. O SDM-Go consegue manter um nível de jogo bastante competitivo, mesmo com a diminuição do desempenho em tabuleiros maiores como o 13X13 e 19X19. Apesar de não superar completamente a estratégia *Prior Knowledge* empregada pelo jogador Fuego, os resultados da utilização da memória SDM apontam para as relevantes contribuições obtidas com o SDM-Go. De fato, a grande desvantagem do SDM-Go é o maior tempo de processamento requerido pelos testes de similaridade executados pela memória SDM. Entretanto, é exatamente este processamento que permite o sucesso da abordagem não supervisionada do SDM-Go. Assim sendo, os resultados alcançados pela implementação do jogador SDM-Go, descrito neste trabalho, indicam a viabilidade de investigar e utilizar técnicas de aprendizado não supervisionadas, como a memória SDM, no tratamento de problemas complexos e de grande espaço de estados, como o jogo de Go. A utilização da memória SDM possibilita substituir abordagens dependentes de domínio, desvinculando o agente de um domínio em específico o que contribui para o reaproveitamento das estratégias de aprendizado não supervisionado em outros problemas. Os resultados obtidos com o SDM-Go mostram que é possível obter um jogador de Go de nível satisfatório com um grau de supervisão inferior ao do agente Fuego. Neste sentido, a memória SDM aparece como uma ferramenta poderosa por permitir que grandes espaços de estados sejam representados por um conjunto reduzido de protótipos, auxiliando no aprendizado do agente.

6.1 Trabalhos Futuros

Apesar do bom desempenho do agente SDM-Go, também foram observadas situações em que a atuação do agente não é satisfatória e requer melhorias. Tais situações se referem, por exemplo, ao caso da identificação do movimento *tesuji*, apresentado na Seção 4.4, onde apenas este movimento garante a vantagem do jogador. Conforme explicado, nestas situações, o critério de similaridade empregado, que utiliza a distância de Hamming, não é adequado para capturar a importância do movimento *tesuji*. Na verdade, ao utilizar a distância de Hamming, o *tesuji* é similar ao mesmo conjunto de protótipos que outros movimentos que não são melhores e assim a avaliação deles não se diferencia significativamente. Em virtude disto, propõe-se como trabalho futuro, a realização de um estudo e análise para introduzir heurísticas apropriadas de forma a otimizar o critério de similaridade através da distância de Hamming. Desta maneira, esta proposta de trabalho futuro tem como objetivo melhorar a maneira como o SDM-Go lida com problemas locais que tem um impacto de longo prazo no jogo. Para melhorar também a representatividade dos protótipos armazenados na memória, como trabalho futuro, os autores pretendem aprimorar a representação baseada em vetores de bits através da inserção de rotinas que

tratam as simetrias do tabuleiro. Desta forma, um dado protótipo, além de ser similar a um certo conjunto de estados de tabuleiro, também será similar as transformações simétricas deste conjunto como rotação, espelhamento e inversão de cores das peças. Logo, esta proposta de trabalho futuro tem como objetivo tornar o SDM-Go ainda mais competitivo ao aumentar a capacidade da memória SDM de realizar casamentos parciais de padrões de estados de tabuleiro. Há ainda outros trabalhos futuros que estendem a investigação iniciada pelo agente SDM-Go, implementado neste trabalho:

- a arquitetura do SDM-Go será transferida para um ambiente de alto desempenho e funcionamento paralelo que irá permitir maior poder de processamento e tempo de execução mais rápido para investigações complementares sobre os tabuleiros 13X13 e 19X19, uma vez que estes domínios possuem um maior espaço de estados;
- será também interessante construir outro agente que substitui a memória SDM por Tabelas de transposição e então realizar uma análise comparativa entre tal agente e o SDM-Go. Desta maneira se tornará possível investigar o *tradeoff* entre a precisão da tabela de transposição e a flexibilidade da memória SDM;
- a atualização dos valores dos protótipos da memória SDM é feita de maneira semelhante à atualização UCT dos valores dos nós da árvore, considerando a taxa de vitórias e a frequência de visitas. Uma vez que os valores armazenados na memória SDM são alterados ao longo da experiência do agente nas simulações da busca, uma possibilidade de investigação futura é a utilização do método das diferenças temporais para ajustar os valores dos protótipos;
- o estudo sobre a integração de métodos de processamento digital de imagens para mapear características locais do tabuleiro que possam ser individualmente analisadas, complementando a visão global oferecida pela representação em bits. Por exemplo, o movimento *tesuji* poderia ser reconhecido como uma característica local e armazenado como um padrão local na memória SDM.

Referências Bibliográficas

- [Aguiar e Julia 2013] Aguiar, M. A. e Julia, R. M. S. (A ser publicado 2013). SDM-Go: an Agent for Go with an Improved Search Process Based on Monte-Carlo Tree Search and Sparse Distributed Memory. In *Computer and Information Technology (CIT), 2013 IEEE 13th International Conference on*.
- [Arneson et al. 2009] Arneson, B., Hayward, R., e Henderson, P. (2009). Wolve wins Hex tournament (13th Computer Olympiad 2008 Beijing). *ICGA Journal*, 32(1):48–53.
- [Auer e Fischer 2002] Auer, P. Cesa-Bianchi, N. e Fischer, P. (2002). Finite-time analysis of the multi-armed bandit problem. *Machine Learning*, (47):235–256.
- [Bouzy e Cazenave 2001] Bouzy, B. e Cazenave, T. (2001). Computer Go: an AI Oriented Survey. *Artificial Intelligence*, 132:39–103.
- [Browne et al. 2012] Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., e Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43.
- [Campos e Langlois 2003] Campos, P. e Langlois, T. (2003). Abalearn: Efficient Self-Play Learning of the game Abalone. In *INESC-ID, Neural Networks and Signal Processing Group*.
- [Caraig 2012] Caraig, L. M. (2012). A New Training Algorithm for Kanerva’s Sparse Distributed Memory. *Computing Research Repository (CoRR)*, abs/1207.5774.
- [Chaslot et al. 2006] Chaslot, G.-B., Saito, J.-T., Bouzy, B., Uiterwijk, J., e Jaap van den Herik, H. (2006). Monte-Carlo Strategies for Computer Go. *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, pp. 83–91.
- [Coulom 2006] Coulom, R. (2006). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Proceedings of the 5th international conference on Computers and games*, pp. 72–83.
- [Dahl 1999] Dahl, F. (1999). Honte, a Go-playing program using neural nets. *Workshop on Machine learning in Game Playing*, pp. 205–223.
- [Denning 1989] Denning, P. (1989). Sparse Distributed Memory. *American Scientist*, (77):333–335.
- [Enzenberger 2003] Enzenberger, M. (2003). Evaluation in Go by a neural network using soft segmentation. *10th advances in computer games conference*, pp. 97–108.

- [Enzenberger et al. 2010] Enzenberger, M., Müller, M., Arneson, B., e Segal, R. (2010). Fuego - An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search. *IEEE Trans. Comput. Intellig. and AI in Games*, pp. 259–270.
- [FSF 2013] FSF (2013). Gnu GO - GNU Project - Free Software Foundation. Disponível em: <<http://www.gnu.org/software/gnugo/>>. Acesso em: 28/07/2013.
- [Gelly e Silver 2011] Gelly, S. e Silver, D. (2011). Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, 175(11):1856–1875.
- [Gelly e Wang 2006] Gelly, S. e Wang, Y. (2006). Exploration exploitation in go: UCT for Monte-Carlo go. In: NIPS-2006: On-line trading of Exploration and Exploitation Workshop. In *In Twentieth Annual Conference on Neural Information Processing Systems (NIPS)*.
- [Gelly et al. 2006] Gelly, S., Wang, Y., Munos, R., e Teytaud, O. (2006). Modification of UCT with patterns in Monte-Carlo go. Technical report, Institut National de Recherche en Informatique et en Automatique.
- [Hely e Hayes 1999] Hely, T.A. Willshaw, D. J. e Hayes, G. (1999). A New Approach to Kanerva's Sparse Distributed Memory. *IEEE Transactions on Neural Networks*, pp. 101–105.
- [Kanerva 1988] Kanerva, P. (1988). *Sparse Distributed Memory*. MIT press.
- [Kierulf 1990] Kierulf, A. (1990). *Smart Game Board: a workbench for game-playing programs with Go and Othello as case studies*. PhD thesis, ETH, Zürich.
- [Kishimoto 2005] Kishimoto, A. (2005). *Correct and efficient search algorithms in the presence of repetitions*. PhD thesis, University of Alberta.
- [Kocsis e Szepesvari 2006] Kocsis, L. e Szepesvari, C. (2006). Bandit based Monte-Carlo planning. *15th European Conference on Machine Learning*, pp. 282–293.
- [Lee et al. 2009] Lee, C.-S., Wang, M.-H., Chaslot, G., Hooock, J.-B., Rimmel, A., Teytaud, O., Tsai, S.-R., Hsu, S.-C., e Hong, T.-P. (2009). The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computer Intelligence and AI in Games*, 1(1):73–89.
- [Levene e Bar-Ilan 2005] Levene, M. e Bar-Ilan, J. (2005). Comparing move choices of chess search engines. *ICGA Journal*, 28(2):67–76.
- [Littman 1994] Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the 11th International Conference on Machine Learning (ML-94)*, pp. 157–163. Morgan Kaufman.
- [Marsland 1997] Marsland, T. (1997). The Anatomy of Chess Programs. *Papers from the 1997 AAAI Work-shop*.
- [Monteiro e Ribeiro 2003] Monteiro, S. T. e Ribeiro, C. (2003). Aprendizagem da Navegação em Robos Moveis a Partir de Mapas Obtidos Autonomamente. In *Anais do XXIII Congresso da Sociedade Brasileira de Computacao*, volume 1, pp. 152–162.

- [Müller 2002] Müller, M. (2002). Counting the score: Position evaluation in computer Go. In *Computers and Games: 4th International Conference, CG 2004*, volume 3846, pp. 273–284. Springer.
- [Müller 2013] Müller, M. (2013). Documentação online Smart Game Library: Monte Carlo Tree Search. Disponível em: <http://fuego.sourceforge.net/fuego-doc-1.1/smartgame-doc/group__sguctgroup.html>. Acesso em: 28/07/2013.
- [Paris et al. 2003] Paris, B. U., Bouzy, B., e Helmstetter, B. (2003). Monte-Carlo Go Developments. In *ACG*, pp. 159–174. Kluwer Academic.
- [Russell e Norvig 2004] Russell, S. e Norvig, P. (2004). *Artificial Intelligence - A Modern Approach*. Campus, 2 edition.
- [Schraudolph et al. 1994] Schraudolph, N. N., Dayan, P., e Sejnowski, T. J. (1994). Temporal Difference Learning of Position Evaluation in the Game of Go. volume 6, pp. 817–824. Morgan Kaufmann, San Francisco.
- [Shannon 1950] Shannon, C. E. (1950). Programming a computer for playing chess. *Philosophical Magazine*, 41(314):256–275.
- [Silver 2009] Silver, D. (2009). *Reinforcement learning and simulation-based search in computer Go*. PhD thesis, University of Alberta.
- [Silver et al. 2007] Silver, D., Sutton, R., e Müller, M. (2007). Reinforcement learning of local shape in the game of Go. *20th International Joint Conference on Artificial Intelligence*, pp. 1053–1058.
- [Silver e Müller 2012] Silver, D. Sutton, R. S. e Müller, M. (2012). Temporal-difference search in computer Go. *Machine Learning*, (87):183–219.
- [Spinelli 2003] Spinelli, J. M. (2003). Error correcting and detecting code. In *Encyclopedia of Computer Science*, pp. 674–677. John Wiley and Sons Ltd., Chichester, UK.
- [Sutton e Barto 1998] Sutton, R. e Barto, A. (1998). *Reinforcement Learning: An Introduction*. MIT press.
- [Thorpe e Anderson 1996] Thorpe, L. e Anderson, C. W. (1996). Traffic light control using sarsa with three state representations. *Technical Report, IBM Corporation*.
- [Walker 2000] Walker, M. A. (2000). An application of reinforcement learning to dialogue strategy in a spoken dialogue system for email. *Artificial Intelligence Research 12*, pp. 387–416.
- [Wu e Meleis 2009] Wu, C. e Meleis, W. (2009). Function Approximation Using Tile and Kanerva Coding For Multi-Agent Systems. *Adaptive Learning Agents (ALA) workshop in 8th Int. Conf. on Autonomous Agents and Multiagent Systems*, pp. 10–15.
- [Yang e Turner 2005] Yang, L., D. A. e Turner, J. A. (2005). *Handbook of Chinese mythology*. ABC-CLIO Ltd.