Go-Ahead: Melhorando heurísticas *Prior-Knowledge* através de informações extraídas das simulações *Play-Out*

Gabriel Machado Santos

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2015

Gabriel Machado Santos

Go-Ahead: Melhorando heurísticas *Prior-Knowledge* através de informações extraídas das simulações *Play-Out*

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Inteligência Artificial

Orientador: Rita Maria da Silva Julia

Uberlândia
2015

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

S237g
2015

Santos, Gabriel Machado, 1985-
Go-Ahead [recurso eletrônico] : melhorando heurísticas Prior-Knowledgeatravés de informações extraídas das simulaçõesPlay-Out / Gabriel Machado Santos. - 2015.

Orientadora: Rita Maria da Silva Julia.
Dissertação (mestrado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.di.2023.7078
Inclui bibliografia.
Inclui ilustrações.

1. Computação. I. Julia, Rita Maria da Silva, 1960-, (Orient.). II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3

Glória Aparecida
Bibliotecária Documentalista - CRB-6/2047

SERVIÇO PÚBLICO FEDERAL
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Ata da defesa de DISSERTAÇÃO DE MESTRADO junto ao Programa de Pós-graduação em Ciência da Computação da Faculdade de Computação da Universidade Federal de Uberlândia.

Defesa de Dissertação de Mestrado Acadêmico: PPGCO- 06/2015

Data: 24 de abril de 2015

Discente: Gabriel Machado Santos Matrícula: 11222CCP004

Título do Trabalho: Go-Ahead: Melhorando heurísticas *Prior-Knowledge* através de informações extraídas das simulações *Play-Out*

Área de concentração: Ciência da Computação

Linha de pesquisa: Inteligência Artificial.

Às 14h30min do dia 24 de abril do ano de 2015 na sala 1B132, Bloco 1B, Campus Santa Mônica da Universidade Federal de Uberlândia, reuniu-se a Banca Examinadora, designada pelo Colegiado do Programa de Pós-Graduação em Ciência da Computação composta pelos professores doutores: Gina Maira Barbosa de Oliveira – FACOM/UFU, Luiz Chaimowicz – DCC/UFMG e Rita Maria da Silva Julia - FACOM/UFU, orientadora do candidato.

Iniciando os trabalhos a presidente da mesa Prof.ª Dr.ª Rita Maria da Silva Julia apresentou a Banca Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir a senhora presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu os conceitos finais.

Em face do resultado obtido, a Banca Examinadora considerou o candidato **aprovado**.

Esta defesa de Dissertação de Mestrado Acadêmico é parte dos requisitos necessários à obtenção do título de Mestre. O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, legislação e regulamentação interna da Universidade Federal de Uberlândia.

Nada mais havendo a tratar foram encerrados os trabalhos às 17 horas e 50 minutos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Prof. Dr. Luiz Chaimowicz
DCC/UFMG

Prof.ª Dr.ª Gina Maira Barbosa de Oliveira
FACOM/UFU

Prof.ª Dr.ª Rita Maria da Silva Julia
FACOM/UFU (Orientadora)

Este trabalho é dedicado ao meu pai José Maria, minha mãe Silene e meus irmãos Gustavo Henrique e Felipe Leonardo.

Agradecimentos

Primeiramente, agradeço aos meus pais, que além de educação me deram todas as condições para o meu pleno desenvolvimento como ser humano.

Agradeço à minha Orientadora, Amiga e Professora, Rita Maria, pelo voto de confiança em meu potencial, por todas as instruções oferecidas, profissionais e pessoais, e pelo compartilhamento de todo seu profissionalismo que possibilitou o desenvolvimento deste trabalho de Mestrado.

Agradeço a todos os professores que já tive, pois além de tutores serviram como inspiração durante toda minha formação.

Agradeço também a todos os meus amigos e amigas, que contribuíram, de alguma forma, para a concretização deste trabalho.

"1. Write down the problem.

2. Think real hard.

3. Write down the solution."

(Feynman, R. ***The Feynman Algorithm****)*

Resumo

Apesar de muito antigo, originado provavelmente na China há 4000 anos atrás, o jogo de Go é um dos maiores desafios na área de Inteligência Artificial. Neste trabalho de Mestrado é descrito o agente Go-Ahead: um jogador automático para o jogo de Go que utiliza uma técnica inovadora a fim de melhorar a acuidade dos valores pré-estimados para movimentos candidatos a serem introduzidos na clássica árvore de busca Monte Carlo (MCTS) utilizada por vários agentes de ponta na cena de *Computer-Go*.

Go-Ahead foi desenvolvido sobre o conjunto de bibliotecas de um desses agentes: o conhecido jogador automático de código aberto Fuego, no qual tais valores pré-estimados são obtidos através de uma heurística chamada *Prior-Knowledge*. Go-Ahead contribui para a redefinição da função que gera tais estimativas através de uma técnica que realiza uma combinação balanceada entre a heurística *Prior-Knowledge* e conhecimento relevante extraído das inúmeras simulações *Play-Out* que são repetidamente executadas durante o processo de busca do algoritmo MCTS.

Com tal estratégia, Go-Ahead provê duas contribuições distintas: primeiramente, aprimora a habilidade do agente no processo de escolha de movimentos apropriados; como segunda contribuição, através do balanço efetuado na combinação do *Prior-Knowledge* com o conhecimento extraído dos *Play-Outs*, - o qual é obtido através de um parâmetro ajustável - provê uma alternativa interessante para a atenuação do caráter supervisionado do processo de pré-avaliação de nós inerente aos agentes baseados no MCTS. Tal ganho é resultado da redução do impacto das heurísticas de *Prior-Knowledge* possibilitado pela inserção de novos conhecimentos recuperados durante a busca.

Os resultados obtidos em torneios contra o agente Fuego confirmam os benefícios e as contribuições oferecidas através desta abordagem.

Palavras-chave: Computer-Go. MCTS. Go. Prior-Knowledge. Play-Outs.

Abstract

Despite being a very ancient game, probably originated in China about 2000 BCE, the game of Go is one of the greatest challenges in the field of Artificial Intelligence. In this thesis is described the agent Go-Ahead: an automatic Go player that uses a new technique to improve the accuracy of the pre-estimated values of the moves which are candidate to be introduced into the classical Monte Carlo Tree Search (MCTS) algorithm used by many current top agents for Go.

Go-Ahead is built upon the framework of one of these agents: the well known open-source automatic player Fuego, in which these pre-estimated values are obtained by means of a heuristic called prior-knowledge. Go-Ahead copes with the task of refining the calculus of these values through a new technique that performs a balanced combination between the prior-knowledge heuristic and some relevant information retrieved from the numerous play-out simulation phases that are repeatedly executed throughout the Monte Carlo search.

With such a strategy, Go-Ahead provides two distinct contributions: first, it enables the agent to enhance the process of choosing appropriate moves. Second, the balancing in the combination of the prior-knowledge and the play-out information - which is obtained by means of an adjustable parameter - represents an interesting alternative to attenuate the supervised character of the calculus of the node evaluations in MCTS based agents, since it allows to reduce the impact of the prior-knowledge heuristic by strengthening the impact of this information.

The results obtained in tournaments against Fuego confirm the benefits and the contributions provided by this approach.

Keywords: Computer-Go. MCTS. Go. Prior-Knowledge. Play-Outs.

Lista de ilustrações

Lista de tabelas

Lista de siglas

AMAF *All Moves As First*

ADM Avaliação Dinâmica de Movimentos

BT *Bradley-Terry*

FPU *First Play Urgency*

GTP *Go Text Protocol*

KGS Kiseido Go Server

MCTS *Monte Carlo Tree Search*

RAVE *Rapid Action Value Estimation*

TT Tabela de Transposição

UCT *Upper Confidence Bounds Applied to Trees*

UCB *Upper Confidence Bounds*

Sumário

Capítulo 1

Introdução

Este trabalho apresenta o jogador automático de Go chamado Go-Ahead. A motivação maior para a aplicação de técnicas da Inteligência Artificial no campo dos jogos reside no fato de tal área oferecer inúmeras características desejáveis que a torna parâmetro para futuras pesquisas em diversas áreas do mundo real, como por exemplo, tomadas de decisões em investimentos financeiros e inserção de inteligência em robôs.

Antes de se definir e aplicar conceitos de Inteligência Artificial é primordial adquirirmos um entendimento básico sobre o que é inteligência. Em "*The relationship between learning and intelligence*" (JENSEN, 1989) de Arthur R. Jensen, um pesquisador renomado na área de inteligência humana, todos os humanos normais possuem os mesmos mecanismos intelectuais. O que diferencia a inteligência de um ser para outro são as condições bioquímicas e fisiológicas, influenciando diretamente na velocidade de raciocínio, memória de curto prazo e na habilidade de se formar e acessar, precisamente, memórias de longo prazo. A partir dessas características o ser é capaz de aprender e utilizar informações para realizar tomadas de decisões no mundo real, otimizando assim as consequências de suas escolhas.

Para os computadores a definição final é a mesma: inteligência é a capacidade de aprender e utilizar informações de forma a otimizar as tomadas de decisões (ENGELBRECHT, 2007). O que diferencia drasticamente a inteligência humana da artificial são os mecanismos de processamento da informação. Um computador dispõe de bastante memória e alta velocidade no processamento de dados, por outro lado sua capacidade de aprendizagem está diretamente relacionada aos mecanismos intelectuais de seus programadores e na representação de dados utilizada para resolução de problemas específicos. Sendo assim, a organização dos mecanismos intelectuais para computadores geralmente é bem diferente daquela utilizada por humanos.

Após a segunda guerra mundial, pesquisadores independentes começaram a investir, em larga escala, tempo na investigação de máquinas inteligentes. O matemático inglês Alan Turing foi um dos pioneiros na área. Em 1947 ele deu uma palestra afirmando que as pesquisas em IA seriam bem mais sucedidas se fossem realizadas através da programação

de computadores ao invés da construção de máquinas.

Para muitos o ano de 1950 marcou o início do desenvolvimento da área de Inteligência Artificial. Em tal ano, Turing apresentou o artigo "*Computing Machinery and Intelligence*"(TURING, 1950), o qual discutia as condições para se considerar uma máquina inteligente, propondo o famoso teste de Turing (TURING et al., 2004), sugerindo que se uma máquina conseguisse se passar por um humano diante de um inteligente observador então ela certamente poderia ser considerada como inteligente. No mesmo ano Claude Shannon definiu as estratégias básicas de um jogador de xadrez (SHANNON, 1950). Em seu artigo, Shannon descreve duas abordagens para a tomada de decisões no jogo: a primeira baseada em técnicas de força bruta no espaço de busca e a segunda baseada na utilização de conhecimentos específicos do domínio, gerando direção na busca através do espaço de estados, possibilitando ao agente otimizar sua busca por melhores jogadas.

Durante o desenvolvimento da área de Inteligência Artificial vários problemas foram abordados e considerados desafios inerentes na construção de agentes inteligentes de qualquer espécie (RUSSELL; NORVIG, 1995), como por exemplo:

- A representação e modelagem formal dos dados relativos ao problema;
- A definição de uma estratégia de planejamento relativa à abordagem do problema;
- O número de agentes envolvidos, cooperativamente ou não, na abordagem do problema;
- A adaptabilidade dos agentes de acordo com mudanças no planejamento inicial;
- O constante aprendizado e identificação de padrões de acordo com os dados processados;
- A capacidade de geração de novas regras(inferência) a partir daquelas previamente definidas;

1.1 Motivação e Objetivos

A área de jogos tem sido amplamente explorada por pesquisadores da área de Inteligência Artificial, tanto por apresentar grandes desafios de evolução quanto por abordar problemas similares aos do mundo real(RUSSELL; NORVIG, 1995). Embora suas regras desfrutem de uma menor formalidade sendo, ao mesmo tempo, menos determinísticas, a elaboração bem sucedida de agentes inteligentes para tal área tem contribuído bastante para a geração de soluções, algumas vezes não ótimas mas satisfatórias, para várias instâncias de problemas da vida real, tais como:

- ❏ Planejamentos autônomos e agendamentos;
- ❏ Diagnósticos médicos;
- ❏ Controles autônomos, como por exemplo visão computacional;
- ❏ Planejamento logístico;
- ❏ Robótica;
- ❏ Processamento de linguagem natural;

Apesar de suas regras simples, o jogo de Go apresenta uma grande complexidade estratégica, o que o torna, dessa forma, consideravelmente mais difícil de se resolver do que o jogo de Xadrez, por exemplo. Existem, relativamente, poucas regras no jogo Go, que são em sua maioria simples e intuitivas. Logo, a complexidade do jogo não vem da quantidade de regras, mas da diversidade de situações que elas desencadeiam em virtude da gigantesca dimensão de seu espaço de busca. A tabela 1 apresenta uma comparação entre alguns jogos de tabuleiros em função de tal quesito.

Tabela 1 – Complexidade do espaço de estados e taxa de ramificação da árvore de alguns jogos de tabuleiro.

Jogo	Ramificação Média da Árvore	log(Espaço de Estados)
Connect Four	4	13
Damas	2.8	18
Gamão	250	20
Xadrez	35	50
Go(19x19)	250	172

Existem várias configurações para o tabuleiro de Go, sendo a mais desafiadora aquela composta por 19 linhas horizontais e 19 linhas verticais, em um total de 361 interseções. O jogo é muito sensível ao movimento das peças, podendo apenas uma jogada errada comprometer todo o resultado final. Geralmente o jogo é dividido em três fases: a abertura, o meio de jogo e o final de jogo. Em cada fase existem técnicas e estratégias especiais destinadas a obter o máximo de proveito para seu jogador. Sendo assim, em cada jogada é muito importante encontrar o que se chama de o "movimento mais forte".

Devido à sua grande complexidade de espaço de estados, os programas mais fortes foram capazes de ganhar de profissionais humanos apenas com várias peças de vantagem (até a data deste trabalho de mestrado). Isso evidencia que as técnicas tradicionais de IA não têm sido fortes o suficiente para levar agentes inteligentes a um nível profissional.

Em 2006, a introdução de um algoritmo utilizando simulações Monte Carlo juntamente com a fórmula *Upper Confidence Bounds Applied to Trees* (UCT)(GELLY et al., 2006a) (KOCSIS; SZEPESVÁRI, 2006) (GELLY; SILVER, 2007) causou um significante avanço no desenvolvimento dos agentes inteligentes. Dentre esses agentes, destaca-se o programa Fuego(ENZENBERGER et al., 2010), um *framework* de código aberto além de um dos programas de estado da arte mais forte atualmente. A *engine* Fuego foi desenvolvida utilizando-se algumas técnicas fortemente supervisionadas, confiando ,principalmente, em heurísticas utilizadas durante as simulações no algoritmo de busca, livro de abertura, contendo os movimentos mais fortes para o início do jogo e heurísticas, que têm por objetivo atribuir um número virtual de vitórias e derrotas a alguns nós na árvore.

No caso do Fuego, o poder do algoritmo de busca utilizado está nas simulações que o mesmo realiza durante a escolha de um movimento. Em cada **episódio** de uma busca simula-se um jogo completo. Um episódio divide-se em duas grandes etapas: construção da árvore e simulação *playout*. A construção da árvore consiste em traçar um caminho definido pelos nós com melhores avaliações. Tal caminho culmina com a inserção de um novo nó. Cada nó inserido na árvore tem seu valor inicial definido por heurísticas chamadas de *prior-knowledge*(caráter supervisionado). O *playout* consiste em simulações de jogadas, a partir do nó inserido na árvore até o final do jogo, definidos por regras heurísticas. Ao final de cada episódio atualizam-se os valores dos nós que pertencem à árvore de busca e fazem parte de tal caminho. Durante o processo de busca, ao inserir um nó na árvore, o algoritmo atribui um número virtual de derrotas e vitórias a tal nó, baseado em observações realizadas em diferentes configurações de tabuleiros. Os nós envolvidos na fase de *playout* não são inseridos na árvore. Apesar disso, o resultado do *playout* (vitória ou derrota) é utilizado para atualizar os valores dos nós da árvore que pertencem, unicamente, ao caminho daquele episódio. Assim sendo, os resultados de um episódio UCT não interferem nos de outros.

Por outro lado, o algoritmo *Rapid Action Value Estimation* (RAVE) (GELLY; SILVER, 2011), baseado na técnica AMAF (BRÜGMANN, 1993), trouxe melhorias às simulações UCT ao permitir que as atualizações de avaliações dos nós da árvore em um dado episódio sejam também consideradas na reavaliação de nós da árvore produzidos a partir das mesmas ações ocorridas em outros caminhos.

O presente trabalho se constitui da criação e implementação do agente Go-Ahead, que tem como base a plataforma Fuego. A idéia principal do agente Go-Ahead é utilizar um algoritmo capaz de extrair o máximo de informações úteis da fase de *playout* em uma busca Monte Carlo, reduzindo o impacto do uso da heurística do *Prior-Knowledge*.

A abordagem a ser desenvolvida deverá atribuir ao agente, de uma forma não supervisionada, a habilidade de extrapolação de um movimento de acordo com o momento corrente no jogo. Por exemplo, é sabido que alguns movimentos têm mais valor na fase de abertura do que no final de jogo. Assim, o agente deverá ser capaz de "aprender"

a ordem de alguns movimentos de forma a realizar as simulações mais acertadamente, atribuindo mais qualidade às simulações e melhorando, de uma forma geral, o poder do jogador automático.

Sendo assim, com o desenvolvimento do agente proposto, o autor tem como objetivos principais:

- ❏ Realizar a análise de uma abordagem diferente na pré-avaliação de um determinado movimento;
- ❏ Comparar a abordagem aqui apresentada com a utilizada pelo agente original, por meio de torneios entre os agentes;
- ❏ Investigar a capacidade de aprendizagem do jogador automático através da observação e processamento dos dados gerados durante a fase de *play-out*;
- ❏ Atenuar o caráter supervisionado do agente original, aumentando assim a "inteligência" intrínseca ao mesmo;

1.2 Hipótese

Através da análise dos algoritmos de IA mais utilizados para Go, estudos exaustivos dos conceitos táticos e estratégicos do jogo, e das heurísticas utilizadas pelo melhores jogadores automáticos, surgiram as seguintes perguntas:

- ❏ Como aprimorar o modelo de pré avaliação utilizado atualmente pelos jogadores?
- ❏ Sendo esse modelo estático, é possível melhorá-lo adicionando nele um caráter dinâmico?
- ❏ A enorme quantidade de dados gerados durante a fase de play-out pode ser aproveitada para a extração de alguma forma de conhecimento relevante?

A partir de tais perguntas, dois problemas foram abordados: O que fazer? A ideia principal foi utilizar os dados gerados na fase de *play-out* para a extração de informações relevantes que possam ser usadas posteriormente em situações similares. Como fazer? Como utilizar esses dados para o aperfeiçoamento das técnica de pré-avaliação dos nós candidatos a inserção na árvore? É possível utilizar um balanço entre a heurística já utilizada e o conhecimento dinamicamente obtido pelo agente?

Todas essas perguntas são respondidas e comprovadas no capítulo 5.

1.3 Contribuições

A contribuição do agente Go-Ahead consiste na criação de um algoritmo capaz de realizar a extração de informações úteis, relacionadas aos movimentos realizados na fase de *play-out*, utilizando, posteriormente, tais informações para aumentar o poder de pré-avaliação de um movimento candidato a ser inserido na árvore de busca durante a fase de expansão.

Tal informação é representada através de, principalmente, dois parâmetros: a frequência com que um movimento é simulado na fase de *play-out* e o reforço (derrota ou vitória) obtido ao final de um *play-out*. Essas informações serão armazenadas progressivamente em uma tabela *hash* à medida que as simulações ocorram.

Através da utilização de tais informações no aprimoramento da técnica de pré-avaliação de um movimento candidato a inserção na fase de expansão, o agente alcançou seus objetivos provendo as seguintes contribuições:

- Aumento da performance do agente, uma vez que os movimentos candidatos a inserção na árvore são melhores pré-avaliados, possibilitando assim a escolha de movimentos mais fortes;
- Apresentação de uma alternativa interessante para a atenuação do caráter supervisionado do agente, uma vez que o impacto da heurística de *prior-knowledge* é diminuído;

Sendo assim, os resultados auspiciosos obtidos pelo agente Go-Ahead em torneios contra Fuego provam que a abordagem proposta representa uma estratégia apropriada para a atenuação da supervisão e melhoria performática do jogador automático.

1.4 Organização da Dissertação

Essa dissertação está organizada da seguinte maneira:

- Capítulo 2, apresenta o referencial teórico utilizado na criação do agente Go-Ahead.
- Capítulo 3, descreve os principais agentes na cena de *Computer Go* relacionados ao trabalho apresentado nessa dissertação.
- Capítulo 4, descreve os detalhes da criação do agente Go-Ahead, apresentando arquitetura utilizada, análises de performance, modo de funcionamento e fluxo de execução.
- Capítulo 5, apresenta os resultados obtidos em torneios realizados contra o agente Fuego (predecessor desse trabalho). Além disso, é realizada uma análise comparativa dos resultados alcançados.

- Capítulo 6, mostra as conclusões diante dos resultados obtidos, as principais contribuições desse trabalho e as próximas atividades a serem desenvolvidas.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta o referencial teórico utilizado para o desenvolvimento do agente Go-Ahead. As seções neste capítulo apresentam conceitos relacionados ao jogo de Go, à busca MCTS e às diferentes políticas relacionadas ao processo.

2.1 O Jogo de Go

Apesar de pouco conhecido no Brasil, o jogo de Go é bastante popular em países como China, Japão e Coréia. Acredita-se que o jogo tenha sido criado há cerca de 4000 anos atrás na China ou Himalaia. O jogo é entendido por muitos como uma guerra por território, onde contam não só suas habilidades táticas e estratégicas mas também sua capacidade de prever as ações de seu oponente. A mitologia, inclusive, conta que o futuro do Tibete uma vez foi decidido sobre um tabuleiro de Go, quando um de seus chefes Budistas recusou ir ao campo de batalha. Ao invés disso ele desafiou o agressor a uma partida de Go com o intuito de evitar o espalhamento de sangue (AN, 2005).

Go é, provavelmente, o mais antigo jogo de tabuleiro no mundo. Acredita-se que o primeiro imperador da China - considerando também uma figura mitológica - inventou o jogo com o intuito de melhorar o raciocínio de seu filho, o qual dispunha de um certo retardo cognitivo. Embora acredita-se que o jogo tenha sido originado na China, foi no Japão que ele começou a ganhar força e popularidade.

Inicialmente, o jogo era disponibilizado apenas aos círculos da realeza, mas gradualmente disseminou-se entre o clero e Samurais. O governo Japonês começou a reconhecer o valor do jogo por volta de 1612, onde foram garantidos várias regalias às melhores famílias jogadoras de Go, resultando na criação de várias escolas do jogo. Dessa forma nos 250 anos seguintes, aproximadamente, a rivalidade intensa entre tais escolas permitiu um avanço significativo no entendimento e forma de jogar. Consequentemente foram elaborados rankings e métricas para o jogo, as quais são comentadas posteriormente nesta seção.

Assim como o Xadrez, Go é um jogo de habilidades. Alguns profissionais e especialistas descrevem o jogo, comparando-o com Xadrez, como se estivesse ocorrendo quatro partidas de Xadrez em um só tabuleiro. O jogo se difere em muitos aspectos do Xadrez. Suas regras são poucas e simples, contudo é um enorme desafio às habilidades analíticas de um jogador, onde um escopo massivo de possibilidades compõe o espaço de busca dificultando, até mesmo para computadores, uma análise mais profunda do jogo.

2.1.1 O que torna o jogo especial

O jogo de Go é um desafio intelectual extraordinário. Apesar de possuir poucas e simples regras, ele oferece um enorme desafio ao campo da Inteligência Artificial. Mesmo os melhores programas atualmente realizam, eventualmente, erros considerados óbvios e inocentes. Além do mais, o jogo oferece diversas vantagens a quem aprecia jogos de habilidades analíticas, tais como:

- O jogo oferece um escopo enorme para análises e experimentos, especialmente na fase de abertura, onde são definidos todos os *frameworks* de jogo, além da estratégia e características táticas. Apesar disso, é possível que um jogador se torne bastante forte sabendo apenas alguns padrões de jogo.
- Outra grande vantagem é seu eficiente sistema de *handicapping*, o qual permite que jogadores de diferentes níveis joguem de uma forma mais balanceada sem distorcer as características do jogo.
- O objetivo em Go é fazer mais território que seu oponente, cercando espaços no tabuleiro ou atacando as peças adversárias eficientemente. Em um tabuleiro grande, como o 19x19, é possível cometer erros em uma área e ainda sim ganhar o jogo.
- De acordo com a grande possibilidade de jogadas, cada jogo acaba desenvolvendo um caráter ímpar. É muito difícil existir repetições de jogo. Assim como um jogador precisa apenas de mais território para ganhar, é muito difícil haver empates no jogo, apesar de uma partida poder se manter balanceada até o final.

2.1.2 Ranking

Embora não padronizado, o nível de um jogador é definido utilizando-se as métricas *Kyu* e *Dan*(SENSEI, 2015b). Os jogadores classificados como *Kyu* são considerados como estudantes do jogo. Os jogadores classificados como *Dan* são considerados mestres no jogo (FINSLAB, 2015). Quando uma pessoa acaba de aprender as regras do jogo ela é considerado 30 *Kyu*. À medida de seu progresso, seu ranking decresce na classificação *Kyu*. Sendo assim a melhor classificação *Kyu* existente é 1 *Kyu*. Se o jogador continuar seu progresso além de 1 *Kyu* ele se torna 1 *Dan* amador, e a partir daí seu *ranking* numérico

aumenta até 9 *Dan* amador(existem algumas variações de um país para outro). A partir daí é possível que um jogador alcance o nível *Dan* profissional.

O sistema de *ranking* adotado mundialmente é exibido na tabela 2:

Tabela 2 – Sistema de ranqueamento em Go.

Classificação	Faixa	Estágio
Dígitos Duplos *Kyu* (DDK)	30-20K	Iniciante
Dígitos Duplos *Kyu* (DDK)	19-10K	Jogador casual
Dígito único *Kyu* (SDK)	9-1K	Amador intermediário
Dan Amador	1-8D	Amador avançado
Dan Profissional	1-9P	Jogador profissional

Embora amplamente usado, esse sistema não possui um padrão universal. Geralmente tal ranqueamento varia de acordo com o país, ou servidor de jogo (SENSEI, 2015a). Ou seja, um jogador considerado 1 *Dan* amador de acordo com um país, pode ser classificado como 2 *Dan* amador ou até 1 *Kyu* em outro.

Durante o desenvolvimento desse trabalho, o autor alcançou o nível de 1 *Dan* amador.

2.1.3 Como jogar

O jogo de Go é jogado por dois jogadores, os quais se alternam em turnos para colocar as peças no tabuleiro, também referido como *Goban*. O tabuleiro mais comumente utilizado é o que contém 19 linhas cruzadas com 19 colunas, resultando em 361 interseções, como mostrado na Figura 1.

Embora o tabuleiro ilustrado pela Figura 1 seja o mais utilizado em jogos de Go, tabuleiros menores, como 9x9 ou 13x13, podem ser utilizados para partidas mais rápidas ou aprendizagem das regras.

A primeira observação é que as pedras são sempre colocadas nas interseções das linhas e colunas e não nas casas como acontece no Xadrez e no jogo de Damas. O jogador que contém as peças pretas sempre começa o jogo. Ele deve colocar uma peça preta em qualquer uma das 361 interseções livres disponíveis no início do jogo. Ao colocar uma peça, essa se torna estática, ou seja, não existe movimentação para tal peça, apenas a possibilidade de remoção da mesma.

Quando os jogadores possuem a mesma força sorteia-se quem pega as peças pretas. Nesse caso, cede-se seis pontos e meio, geralmente, ao jogador das peças brancas com o intuito de contrabalancear a vantagem de se começar. Tal vantagem é também referida como *Komi*.

No caso de jogadores de níveis diferentes, coloca-se pedras de vantagem, ou seja, o jogador mais fraco tem a vantagem de começar com algumas peças a mais já postas no tabuleiro. Tal número de peças pode variar, oficialmente, entre uma e nove peças.

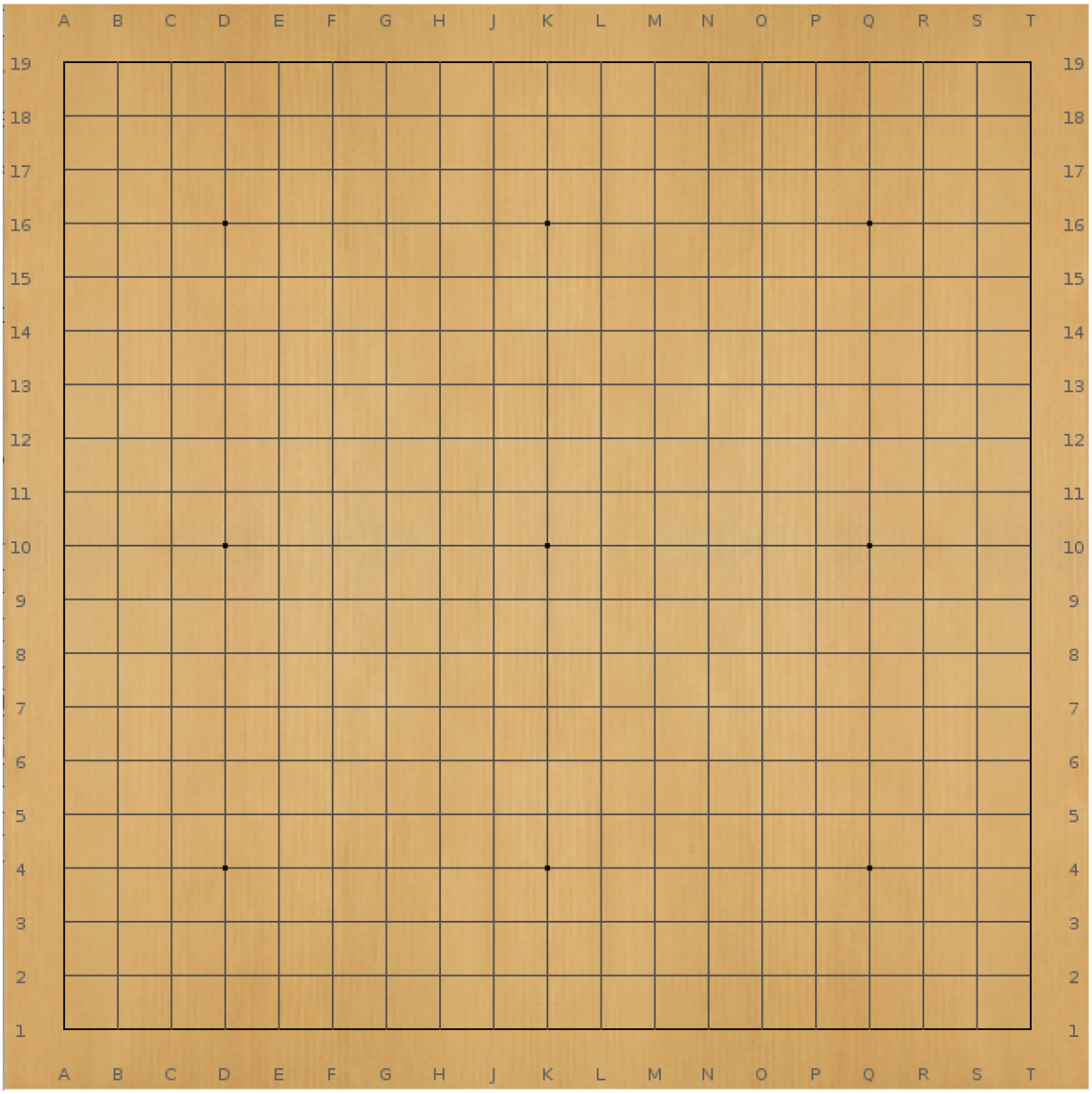

Figura 1 – Tabuleiro de Go vazio com 19 linhas e 19 colunas. O tabuleiro mais comumente utilizado no jogo de Go.

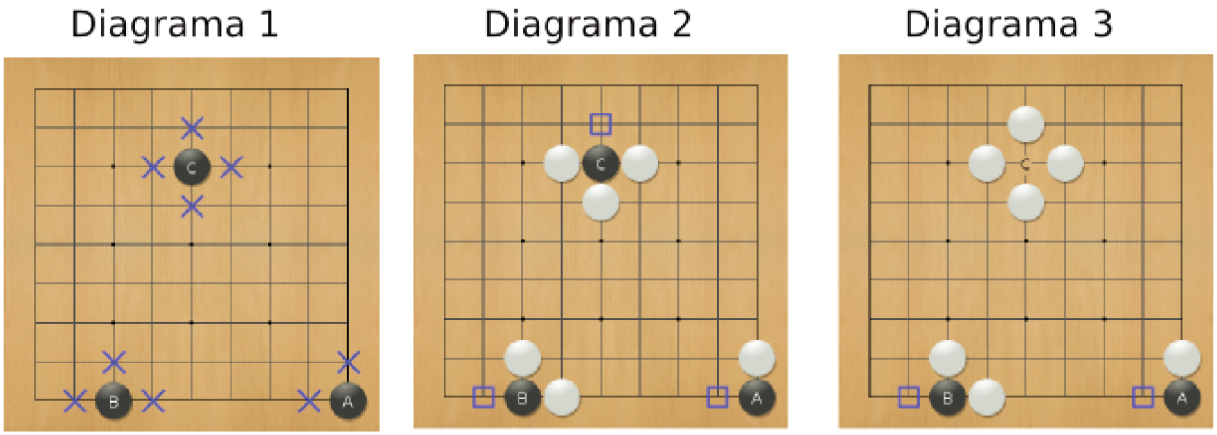

Figura 2 – Diagramas com exemplos de liberdades e capturas.

O objetivo final do jogo é fazer mais território que seu oponente através do cercamento do maior número de interseções no tabuleiro, além da captura de peças e grupos adversários pela remoção de suas liberdades. A seguir são descritos alguns conceitos essenciais ao entendimento do jogo.

2.1.3.1 Capturando Peças e Contando Liberdades

Os pontos (interseções) livres que se encontram vertical ou horizontalmente adjacentes a uma peça, ou um grupo, são chamados de liberdades. Um peça isolada ou um grupo de peças é capturado quando todas as suas liberdades são ocupadas por peças adversárias.

No Diagrama 1 da Figura 2 são exibidas três peças pretas isoladas no tabuleiro. As marcas em X representam as liberdades de cada uma das peças. Repare que a peça A no canto do tabuleiro tem apenas duas liberdades.

Já no Diagrama 2 da Figura 2 as mesmas três peças são exibidas, mas agora com apenas uma liberdade cada uma. A essa situação dá-se o nome de *Atari*, ou seja, quando uma peça, ou grupo, tem apenas uma liberdade restante. Caso essa liberdade seja ocupada por uma peça adversária, a peça que agora não tem mais liberdades é removida do tabuleiro e contada como ponto para o adversário, como mostrado no Diagrama 3 da figura.

No jogo de Go, é proibido colocar uma peça de tal modo a deixá-la sem liberdade nas jogadas seguintes. Uma peça só pode ser colocada onde não há liberdades caso ela resulte em captura das peças adversárias.

2.1.3.2 Grupos

Peças da mesma cor ocupando pontos adjacentes uma a outra constituem um sólido e conectado grupo. Na Figura 3, as peças pretas constituem um grupo com doze liberdades, as peças brancas na parte inferior do tabuleiro constituem outro grupo também com doze

liberdades e as duas peças brancas marcadas com um círculo não constituem um grupo, já que não são adjacentes uma a outra.

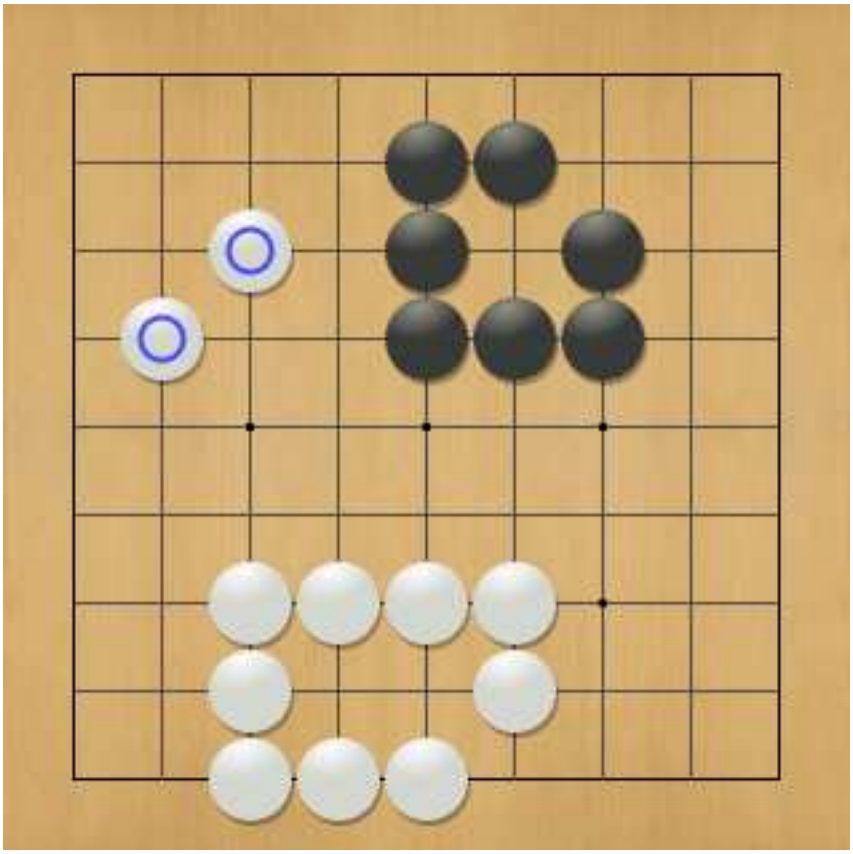

Figura 3 – Exemplos de formações de grupos.

2.1.3.3 Capturando Grupos

Assim como uma peça isolada, um grupo de peças é tratado como uma única entidade. Sendo assim, analogamente à captura de uma peça isolada, um grupo é capturado quando todas suas liberdades são removidas.

A Figura 4 elucida bem esse processo. O Diagrama 1 na figura mostra dois grupos cercados por peças adversárias. O grupo de peças pretas representado na parte de cima do tabuleiro desfruta de apenas uma liberdade central H. Logo, se o jogador das peças brancas realizar uma jogada em H, todo o grupo preto é capturado, assim como mostrado no Diagrama 2.

Por outro lado, o grupo de peças brancas na parte inferior do tabuleiro (Diagramas 1 e 2) contém duas liberdades. O jogador preto não pode jogar em nenhuma das liberdades do grupo branco, indicadas por $E1$ e $E2$, pois isso resultaria em suicídio, o que não é permitido no jogo de Go. Como mencionado anteriormente, tal jogada só seria permitida se resultasse em uma captura imediata do grupo branco, o que não acontece nesse caso devido a outra liberdade do grupo. Logo, tais liberdades são chamadas olhos, e o grupo

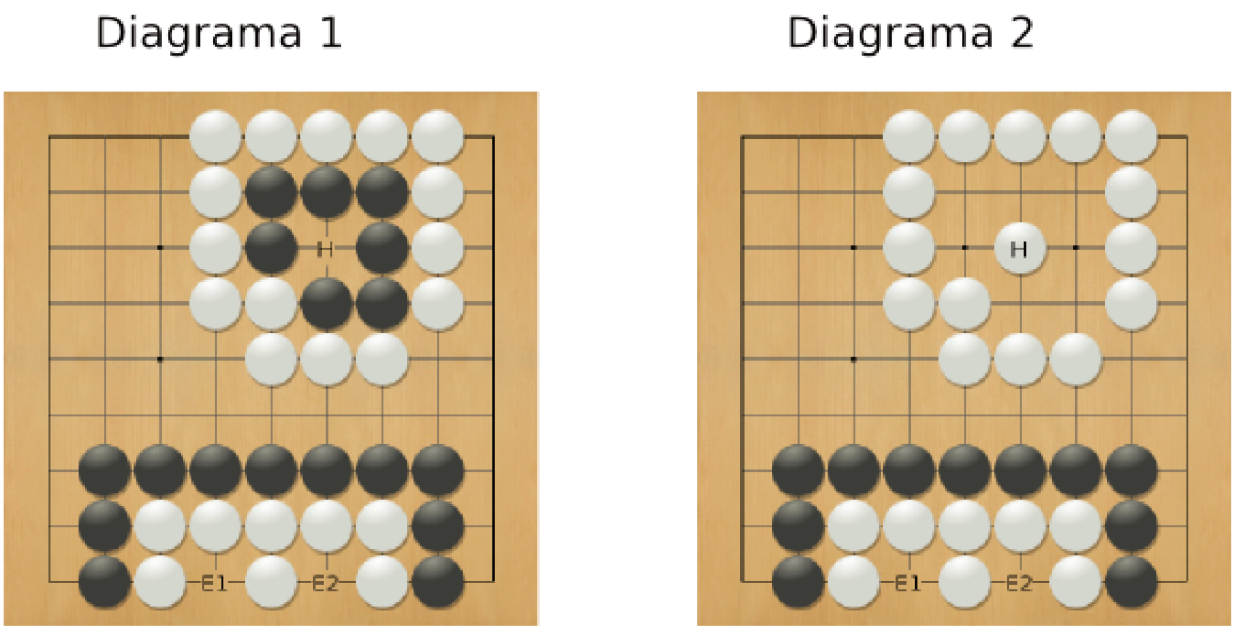

Figura 4 – Exemplos de capturas de grupo e conceitos de vida e morte.

branco é considerado incondicionalmente vivo. Um grupo é considerado incondicionalmente vivo caso tenho no mínimo dois olhos.

2.2 Simulação Monte Carlo

Em alguns sistemas descritos por modelos matemáticos, a obtenção de avaliações analíticas pode ser uma tarefa extremamente complexa devido à natureza de tais modelos. Em sistemas envolvendo buscas em árvores, muitos algoritmos dependem de boas funções avaliativas para os estados representados por nós na árvore. Nos casos em que tais funções não existem ou simplesmente não retornam uma boa aproximação do valor real de tais estados, a simulação computacional pode ser considerada uma ferramenta de grande valia na obtenção de uma avaliação posicional.

Em jogos, uma simulação Monte Carlo é uma forma de se avaliar as ações candidatas a ser tomadas partindo de um estado raiz s_0, a partir do qual se realizam tais ações. Em sua forma mais básica, uma simulação Monte Carlo parte de um estado s_0 selecionando ações de uma forma pseudo aleatória, em um processo de auto-disputa, até que o fim de jogo seja alcançado. Ao final de cada simulação i é retornada uma recompensa R_i indicando o desempenho do agente em tal simulação. O valor atribuído a uma ação a_x, representando nesse caso a probabilidade de vitória do agente inteligente ao escolher tal ação, é estimado pela média aritmética das recompensas obtidas ao final de todas as simulações partindo do estado s_0 tomando a ação a_x. Sendo assim, considerando $Q_n(s_0, a_x)$ o valor estimado da ação a_x a partir do estado s_0, após n simulações, tem-se que:

$$Q_n(s_0, a_x) \leftarrow \frac{1}{n} \times \sum_{i=1}^{n} R_i \tag{1}$$

Através da Lei dos Grandes Números (WASSERMAN, 2004) pode se verificar que na medida em que n tende ao infinito (i.e, $n \to \infty$), o valor estimado $Q_n(s_0, a_x)$ tende ao seu valor real (i.e, $Q_n(s_0, a_x) \to \mu$, onde μ representa o valor real). O valor de $Q_n(s_0, a_x)$ quando $n \to \infty$ será aqui representado por $Q_\infty(s_0, a_x)$. Logo, pode-se dizer que $Q_\infty(s_0, a_x) \to \mu$.

O Teorema do Limite Central (WASSERMAN, 2004) afirma que quando o tamanho da amostra aumenta, a distribuição amostral de sua média aproxima-se cada vez mais de uma distribuição normal. Assim, considerando σ o desvio padrão da recompensa, tem-se que para um n suficientemente grande, $Q_n(s_0, a_x)$ tende a obedecer uma distribuição normal com média $Q_\infty(s_0, a_x)$ e desvio padrão $\frac{\sigma}{\sqrt{n}}$. Dessa forma, conclui-se a seguinte aproximação:

$$Q_n(s_0, a_x) \in [Q_\infty(s_0, a_x) - \frac{\sigma}{\sqrt{n}}, Q_\infty(s_0, a_x) + \frac{\sigma}{\sqrt{n}}] \tag{2}$$

para um n suficientemente grande.

Em sua forma mais básica, uma simulação Monte Carlo é utilizada apenas para avaliar ações a partir de determinados estados. A política que define as ações a serem tomadas durante a simulação pode assumir diversos tipos de estratégia, desde aleatórias até mais determinísticas.

Para facilitação do entendimento, $Q_n(s_0, a_x)$ será referido como $Q(s, a_x)$ no restante do trabalho.

2.3 Busca em Árvore Monte Carlo

O algoritmo de busca em árvore MCTS (CHASLOT, 2010a) (CHASLOT, 2010b) (BRÜGMANN, 1993) (GELLY et al., 2006b), é essencialmente bastante simples. Ele consiste em construir, iterativamente, uma árvore de busca baseado em simulações Monte Carlo. Tal estrutura de dados, por sua vez, é utilizada por agentes inteligentes em uma tentativa de realizar decisões ótimas em problemas complexos como, por exemplo, na área de planejamento em jogos combinatórios.

O algoritmo combina uma estimativa de valor posicional a partir de simulações aleatórias com um método de *best-first search* (KORF, 1993) de busca em árvore. Em sua estrutura mais básica, cada nó pertencente à árvore representa uma posição, estado ou até mesmo uma ação a ser tomada no jogo (MÜLLER, 2002). Esse nó, geralmente, carrega no mínimo duas informações, a saber:

- O valor atual v_i, geralmente representando a média de vitórias obtidas em jogos simulados que passaram por esse nó.

- O número n_i de vezes que essa posição foi visitada.

Com base em tais dados, o algoritmo MCTS seleciona o melhor filho do nó corrente ao final de uma busca.

O processo de construção da árvore é recursivo. Em cada ciclo da recursão um novo nó é inserido na árvore. Dessa forma, assim como ilustrado através das Figuras 5 e 6, cada episódio de busca começa sempre do nó raiz e durante o processo passa por quatro fases que dividem o algoritmo:

1. **Seleção**: essa fase consiste em estipular um caminho utilizando uma política de seleção de nós, partindo do nó raiz, contendo sempre os melhores filhos de cada nó visitado. Esse processo é recursivo até que se encontre um nó folha. A seção 2.5 descreve exemplos de políticas de seleção em árvores.

2. **Expansão**: Uma vez que um caminho foi estipulado no processo anterior e um nó folha foi alcançado, inicia-se o processo de expansão da árvore de busca. No momento da expansão todas as ações possíveis, a partir de tal estado, são consideradas como nós filhos do nó corrente. A partir daí o algoritmo deve escolher o melhor filho para realizar uma simulação até o final de jogo. Como todos os nós possuem o mesmo valor, tal escolha, na forma mais básica do algoritmo, é realizada aleatoriamente. Em alguns casos é configurado um valor $+\infty$ para cada ação inserida na árvore, assim o algoritmo explorará cada nó pelo menos uma vez, como é mostrado na técnica *First Play Urgency* (FPU) (GELLY et al., 2006a) (GELLY; WANG, 2006). Já em outros trabalhos é realizada uma pré-inicialização do nó, com valores retirados de heurísticas (GELLY; SILVER, 2011) (DOMSHLAK; FELDMAN,).

3. **Simulação *Play-out***: esse processo inicia-se a partir do nó inserido no passo anterior. Ele é responsável por determinar e simular movimentos, de acordo com uma política de simulação, até o final de jogo. Ao final de cada simulação i obtém-se uma recompensa R_i.

4. **Retro-propagação**: a partir daí, é iniciada a fase de Retro-Propagação, a qual é responsável por atualizar apenas os nós contidos no caminho determinado na fase de seleção. É importante observar que os nós utilizados nos passos da simulação não são atualizados e, ao invés disso, são descartados ao fim do processo.

A figura 5 ilustra uma representação resumida com as quatro fases do algoritmo MCTS.

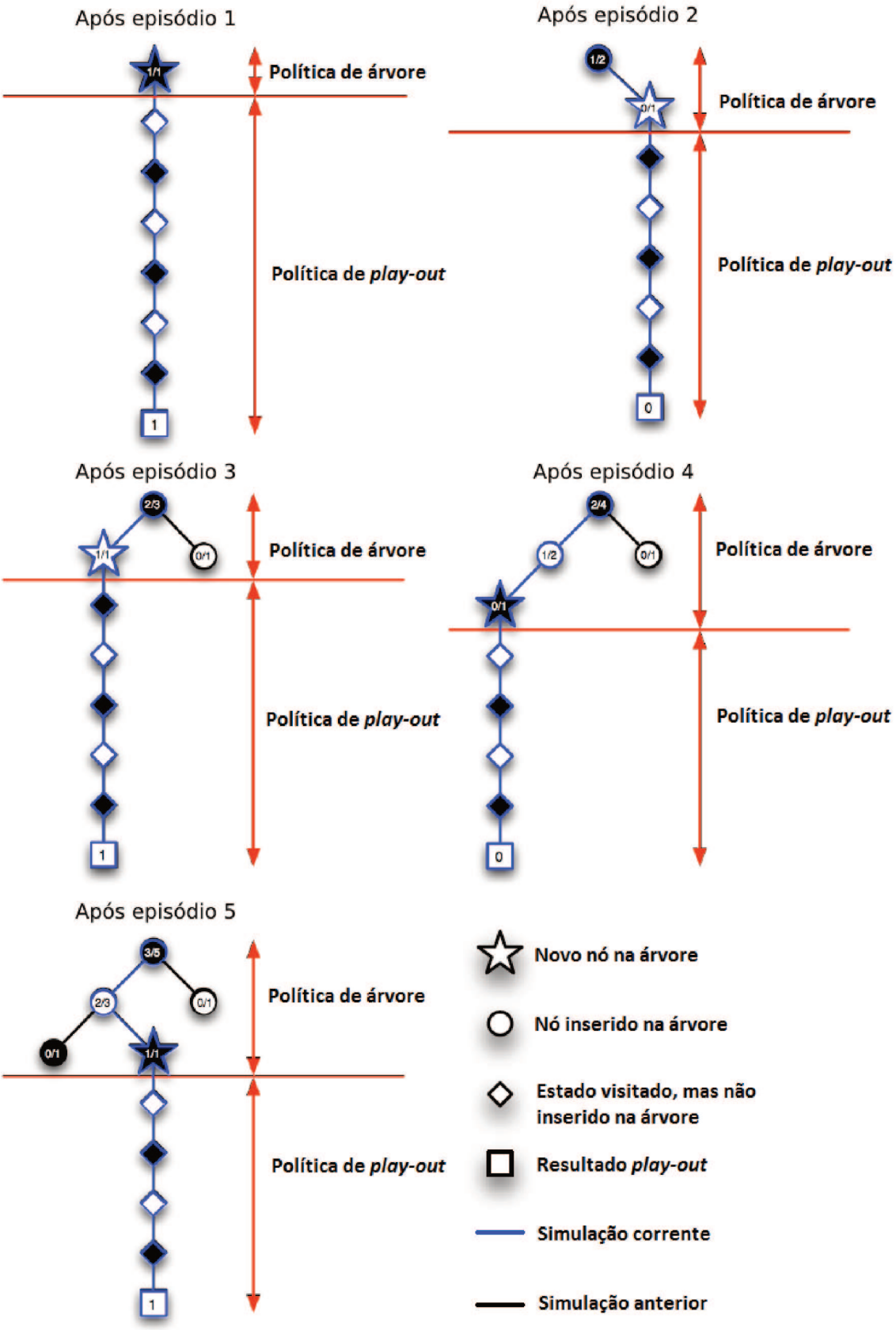

Figura 5 – Resumo dos passos de um episódio da busca Monte Carlo.

Como mencionado anteriormente, a árvore de busca é construída iterativamente. Simultaneamente, os valores estimados de seus nós são progressivamente atualizados com o decorrer dos episódios, de forma que quanto mais episódios melhor a acuidade de tais nós.

Um pseudo-código do algoritmo é dado no algoritmo 1.

No pseudocódigo do algoritmo 1 as linhas 3 a 6 representam o processo de seleção. A constante CNA representa o conjunto de nós pertencentes à árvore de busca. $Seleciona(no\_atual)$ é o método responsável por selecionar, baseado em uma política de busca em árvore, o melhor filho do nó representado pela variável no_atual. O laço iniciado na linha 3 só irá parar quando a variável no_atual receber um nó que não pertence ao conjunto de nós da árvore de busca, ou seja, alcançou o filho de um nó folha.

A linha 8 do algoritmo representa o processo de expansão da busca. Nesse ponto, um dos nós filhos do nó folha alcançado no passo anterior e referenciado pela variável $no\_anterior$, será inserido na árvore de busca.

A linha 10 indica a execução do processo de simulação de um jogo completo iniciado a partir do nó inserido no passo anterior. Uma recompensa $R \in [-1, 1]$ será emitida ao final de tal passo.

O processo de Retro-Propagação inicia-se no *loop* da linha 13. A função $Atualiza(no\_atual, R)$ atualiza o nó referenciado pela variável no_atual com a recompensa R obtida no passo anterior. O *loop* irá terminar apenas quando o nó raiz for atualizado.

Algoritmo 1 Resumo de um episódio MCTS.

Entrada: Nó raiz

Saída: Melhor movimento encontrado

```
1  início
2      %1- Seleção: construindo um caminho através da árvore. %
3      enquanto no_atual ∈ CNA faça
4          no_anterior ← no_atual;
5          no_atual ← Seleciona(no_atual);
6      fim
7      %2- Expansão: adicionando um nó à árvore. %;
8      no_anterior ← Expande(no_anterior);
9      %3- Simulação: simulando um jogo inteiro a partir do nó inserido. %;
10     R ← SimulaJogo(no_anterior);
11     %4- Retro-Propagação: atualizando os valores da avaliação dos nós da árvore estipu-
       lados no passo de Seleção. Será atualizado cada nó, partindo do último adicionado
       até o nó raiz. %;
12     no_atual ← no_anterior;
13     enquanto no_atual! = NULL faça
14         Atualiza(no_atual, R);
15         no_atual ← no_atual.parent;
16     fim
17 fim
```

Algumas vantagens do algoritmo MCTS são listadas a seguir:

- Sem heurística necessária: o algoritmo não exige nenhum conhecimento tático ou estratégico relativo ao domínio para realizar uma decisão.
- Assimétrico: O algoritmo contempla o crescimento assimétrico da árvore. O foco de desenvolvimento e exploração se mantém nos ramos mais interessantes. O imenso espaço combinatório geralmente causa problemas aos algoritmos padrões de busca, tais como os de busca em profundidade ou busca em largura (CORMEN et al., 2001).
- Sem restrição de tempo: o algoritmo pode ser parado a qualquer momento para retornar um movimento.
- Estruturas reutilizáveis: A árvore gerada pode ser mantida ou apagada para buscas subsequentes.
- Fácil implementação: por possuir uma dinâmica de fácil compreensão, sua implementação se torna relativamente simples.

A criação de bons agentes inteligentes para o jogo de Go é uma tarefa considerada extremamente complexa. Apesar das vantagens do algoritmo MCTS listadas acima, ele ainda pode falhar em achar os melhores movimentos mesmo em jogos de complexidade média, como o Go 9x9, devido à grandeza do espaço de busca. Sendo assim, muitas vezes os nós contendo os melhores movimentos não são suficientemente visitados para obter-se um valor confiável. Isso exige que o algoritmo execute muitos episódios de busca até convergir a uma boa solução. Por exemplo, uma boa implementação de Go pode exigir milhares, ou até milhões, de episódios com ajuda de heurísticas para realizar um movimento considerado *expert*.

A Figura 6 ilustra uma simples sequência de cinco passos do algoritmo MCTS em uma árvore de ramificação dois, para facilitar o entendimento. Cada simulação resulta em uma recompensa de 1 para a vitória de preto ou 0 para a vitória de branco (representada no quadrado ao final da árvore). Em cada passo o nó estrela é adicionado à árvore de busca. Ao final de cada simulação *Play-Out* o valor dos nós pertencentes ao caminho selecionado (estrela e círculos) são atualizados de modo a conter o número de vitórias das peças pretas e o total de visitas àquele nó (vitórias/visitas).

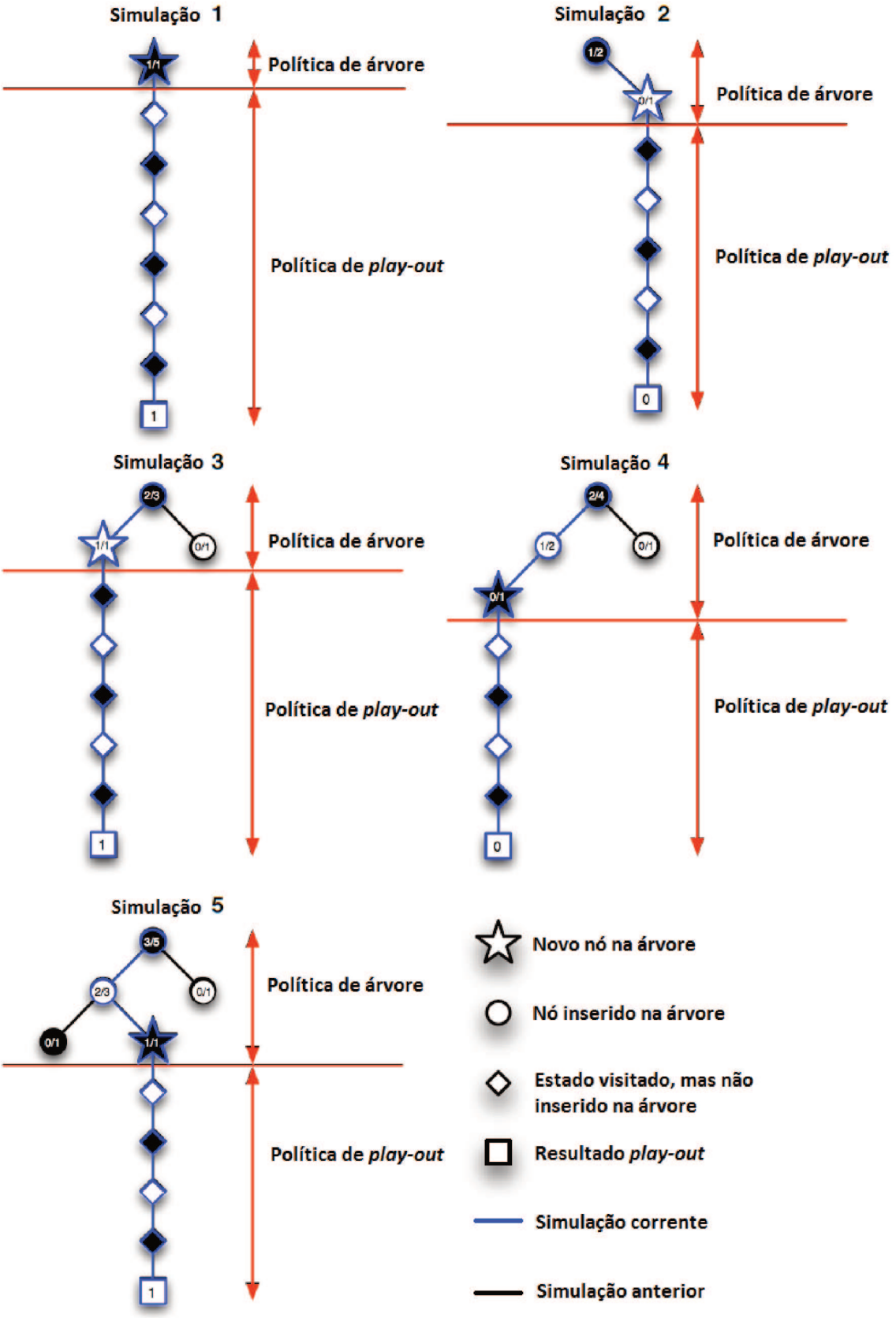

Figura 6 – Exemplo de 5 passos da construção de uma árvore de busca Monte Carlo.(Imagem utilizada com permissão)(GELLY; SILVER, 2011)

2.4 *All Moves As First* (AMAF): Uma Alternativa ao Processo de Atualização no Algoritmo MCTS

Como mostrado na seção 2.3, durante o processo de Retro-Propagação do MCTS original, ao final de cada episódio, os nós atualizados são apenas aqueles que compõem o caminho selecionado no passo Seleção de cada episódio, não afetando assim nós pertencentes a outros ramos da árvore. Diferentemente, a técnica AMAF (HELMBOLD; PARKER-WOOD, 2009) (GELLY; SILVER, 2011), ao término do *play-out*de cada i-ésimo episódio da busca corrente, efetua atualizações de valores que podem extrapolar o caminho definido por tal episódio, conforme mostrado a seguir. O valor AMAF $\tilde{Q}(s, a_x)$ representa a média de todas as simulações em que a ação a_x foi selecionada em **qualquer** turno depois de encontrado o estado s. Isso é representado através da equação 3.

$$\tilde{Q}(s, a_x) \leftarrow \frac{1}{N(s, a_x)} \times \sum_{i=1}^{N(s)} [\tilde{I}_i(s, a_x) \times R_i] \tag{3}$$

Na equação, $N(S, a_x)$ representa o número de vezes em que a ação a_x foi selecionada a partir do estado S na busca corrente, $N(S)$ indica quantas vezes o estado S foi visitado na busca corrente, independente de qual ação foi tomada a partir daí e $\tilde{I}_i(S, a_x)$ é uma função indicadora que retorna 1 caso o estado S tenha sido encontrado em qualquer passo t do $i - ésimo$ episódio, e a ação a_x tenha sido selecionada em qualquer passo $u \geq t$ do mesmo episódio; ou 0 caso contrário. É importante salientar que ambas jogadas, Pretas e Brancas, são consideradas ações distintas, mesmo se jogadas na mesma interseção.

A Figura 7 ilustra uma comparação entre a técnica AMAF de atualização e o processo comumente utilizado no MCTS.

Na Figura 7 é ilustrada uma situação comparando os valores dos movimentos a e b executados pelo jogador preto a partir do estado s, com os resultados mostrados dentro dos quadrados na parte inferior. Sendo assim, executar o movimento a imediatamente após s resulta em duas derrotas, ou seja 0 vitórias em 2 execuções (0/2).Assim, a estimativa Monte Carlo favorece o movimento b com 2 vitórias em 3 execuções (2/3). No entanto, a execução do movimento a em **qualquer** passo subsequente ao estado s resulta em 3 vitórias através de 5 execuções, e a execução do movimento b nas mesmas condições resulta apenas em 2 vitórias sobre 5 execuções. Neste caso, a estimativa AMAF favorece o movimento a. É importante notar que a execução do movimento a a partir do nó raiz não pertence à subárvore $\tau(s)$ e, desse modo, não contribui para a estimativa AMAF $\tilde{Q}(s, a)$.

2.5 Políticas de Seleção em Árvore

Esta seção apresenta as principais políticas de árvore utilizadas no processo de Seleção do algoritmo MCTS referente ao escopo de Go.

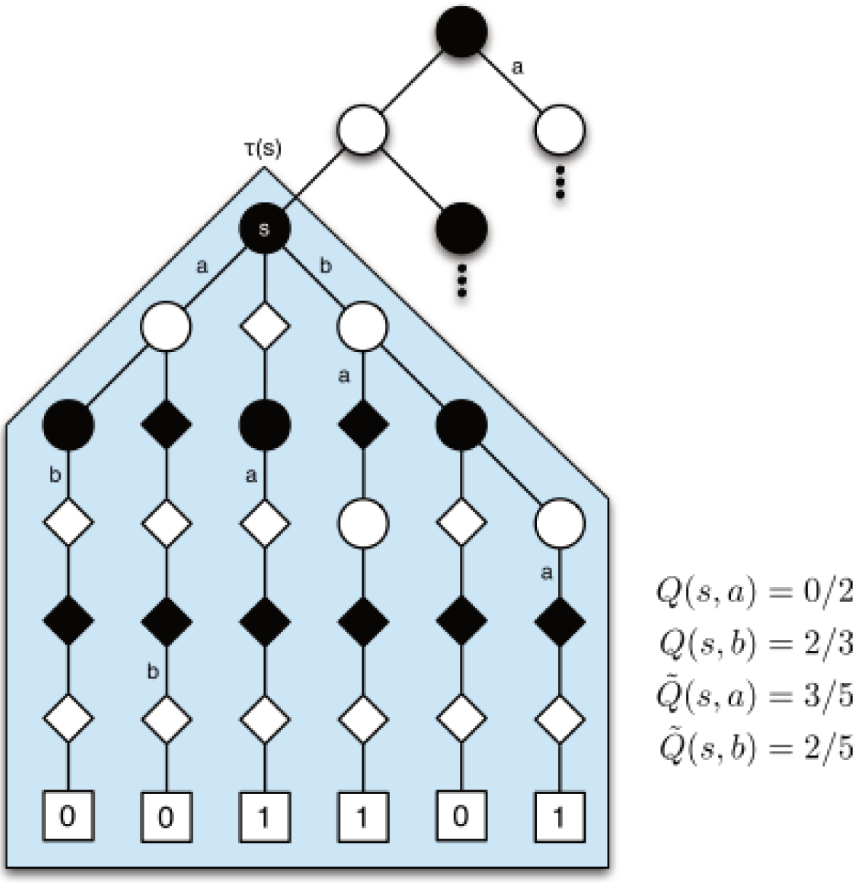

Figura 7 – Comparação entre as estimativas MCTS e AMAF.

2.5.1 *Upper Confidence Bounds Applied to Trees (UCT)*

Um dos métodos mais utilizados para se definir um caminho na árvore de busca durante o processo de Seleção é o *Upper Confidence Bounds* (UCB) (KOCSIS; SZEPESVÁRI, 2006),o qual sugere a inserção de um limite superior ao intervalo de confiança da estimativa MCTS $Q(s,a)$. Em 2006 foi proposta a utilização do método UCB combinado com o processo de seleção na árvore de busca MCTS, resultando no método UCT. Sua principal ideia é utilizar a técnica UCB em cada passo da Seleção, sempre escolhendo o nó que maximize a seguinte expressão:

$$Q^{\oplus}(s,a_x) \leftarrow Q(s,a_x) + (c \times \sqrt{\frac{\ln N(s)}{N(s,a_x)}}) \tag{4}$$

onde $Q^{\oplus}(s,a_x)$ representa o valor UCT, $Q(s,a_x)$, $N(s)$ e $N(s,a_x)$ já foram previamente introduzidos e c representa uma constante de exploração manualmente definida, geralmente 0.7.

A principal vantagem dessa estratégia é o balanço introduzido entre os conceitos *exploration* (tendência a explorar novas regiões da árvore de busca) e *exploitation* (tendência em

manter a busca em regiões melhores avaliadas da árvore) encontrados durante o processo de busca.

2.5.2 *Rapid Action Value Estimation (RAVE)*

O algoritmo RAVE modifica a abordagem MCTS através da utilização da heurística AMAF, aumentando assim o conhecimento extraído de uma simulação *Play-Out* ao custo da inclusão de conhecimentos tendenciosos ou até mesmo menos relevantes (GELLY; SILVER, 2011). Através da combinação do MCTS com a heurística AMAF, o algoritmo RAVE compartilha informações estatísticas entre as subárvores da árvore de busca durante sua execução.

Cada estado s pertencente à árvore de busca é a raiz de uma subárvore. Sendo assim, se durante a Seleção um estado s é visitado, todos os estados subsequentes a s fazem parte da subárvore com raiz em s.

A ideia principal do algoritmo RAVE é generalizar sobre subárvores. A assunção aqui é que uma ação a_x partindo do estado s tenha valor similar partindo de qualquer estado pertencente à subárvore com raiz em s. Dessa forma o valor de a_x é estimado partindo de todas as simulações iniciadas em s, independente da profundidade na árvore em que ela foi selecionada.

Quando o valor AMAF é utilizado para selecionar uma ação a_x partindo do estado s, a ação com maior valor RAVE é selecionada. Dessa forma temos:

$$Rave_a(s) \leftarrow arg\,max_B \tilde{Q}(s, B) \tag{5}$$

onde $Rave_a(s)$ representa a ação selecionada pelo algoritmo RAVE no estado s, B representa um conjunto de todos os nós filhos de s e $arg\,max_B \tilde{Q}(s, B)$ retorna um dos filhos dentre o conjunto B que possui o maior valor AMAF $\tilde{Q}$.

A política RAVE é similar à heurística histórica utilizada na busca Alfa-Beta (SCHAEFFER, 1989). Analogamente ao RAVE, o método de heurísticas históricas se lembra do sucesso de cada movimento em várias profundidades, assim, os movimentos mais bem sucedidos são simulados primeiro em posições subsequentes.

2.5.3 *MC-RAVE*

O algoritmo RAVE rapidamente aprende como generalizar o valor de um movimento específico. Muitas vezes esse valor é estimando erroneamente devido ao fato de que tal movimento pode assumir valores completamente diferentes ao longo da árvore, contrariando a assunção de tal método.

Dessa forma, nas fases iniciais da busca, onde poucos episódio foram executados, a técnica RAVE produz uma melhor estimativa dos movimentos em questão. Contudo, com

o aumento do número de episódios durante o processo, a estimativa Monte Carlo supera a RAVE em termos de acuidade.

Logo, o algoritmo MC-RAVE (GELLY et al., 2012) (GELLY; SILVER, 2011) altera o MCTS através da introdução de um balanço entre as estimativas AMAF e Monte Carlo, assim como mostrado a seguir:

$$Q_\star(s, a_x) \leftarrow [(1 - \beta(s, a_x)) \times Q(s, a_x)] + [\beta(s, a_x) \times \tilde{Q}(s, a_x)] \tag{6}$$

onde $Q_\star(s, a_x)$ é o valor MC-RAVE, $\beta(s, a_x)$ é uma constante de balanço para o estado s e a ação a_x. Sendo assim, quando poucos episódios foram executados, o peso da estimativa AMAF é maior com $\beta(s, a_x) \approx 1$. Contrariamente, quando vários episódios foram executados, o peso da estimativa Monte Carlo é maior com $\beta(s, a_x) \approx 0$.

Analogamente às outras políticas de árvore apresentadas previamente, em cada passo seleciona-se aquele nó que maximiza o valor $Q_\star$ relativo ao estado s.

Capítulo 3

Estado da Arte

3.1 MoGo

O agente MoGo é um dos programas de ponta em *Computer Go* desenvolvido por um grupo de pesquisadores franceses do *Institut National de Recherche en Informatique et en Automatique*(INRIA) (GELLY et al., 2006a) (FR, 2015) (LEE et al., 2009). Tal agente foi o primeiro a utilizar os método Monte Carlo em conjunto com UCT. Ele utiliza diversos padrões durante a execução das simulações *Play-Out* e várias melhorias aplicadas à técnica UCT em conjunto com a busca na árvore.

O agente é dividido em duas partes principais:

- Parte de busca em árvore: aqui é utilizada uma versão otimizada da técnica UCT através da introdução de uma estrutura dinâmica de árvores como citado em (GELLY et al., 2006a) com o intuito de economizar memória.
- Parte de simulações randômicas: onde são realizados os *Play-Outs* em simulações até o final de jogo, descartando os movimentos executados e atualizando os nós da árvore com o resultado obtido.

Em 2009, durante um torneio em Taiwan, o agente venceu pela primeira vez, um jogador profissional utilizando uma vantagem de apenas sete peças (LEE et al., 2009). Com isso, o agente demonstrou o resultado de três avanços científicos e tecnológicos:

- Provou que com a utilização do algoritmo MCTS é possível explorar o imenso espaço de estados do jogo de Go. Isso, de certa forma, revolucionou a noção de planejamento em Inteligência Artificial.
- A avaliação das posições é baseada no algoritmo MCTS, simulando um jogador estocástico de baixo nível sem nenhuma influencia externa.
- O alto grau de paralelismo alcançado (notavelmente com o GRID'5000 (GRID5000, 2015)), aumentou expressivamente o poder de precisão nas avaliações Monte Carlo.

Tais desenvolvimentos mostraram a possibilidade de novas aplicações do algoritmo. Além do jogo de Go, o agente MoGo é baseado em tecnologias inovadoras que podem ser utilizadas em diversos campos, como por exemplo, economia de recursos, o que é crucial em problemas ambientais.

3.2 CrazyStone

CrazyStone é um programa jogador de Go, desenvolvido por Rémi Coulom, que faz uso da busca em árvore Monte Carlo em conjunto com uma técnica Bayesiana de aprendizagem baseada no modelo *Bradley-Terry* (BT) (COULOM, 2007b). Nesta técnica, cada movimento é representado por um conjunto fixo de características, em que cada uma descreve uma propriedade específica do movimento candidato que definirá a próxima jogada. Tal representação é baseada em dez características distintas.

O agente CrazyStone, além de contar com uma política de simulação baseada na modelo BT, conta também com uma Tabela de Transposição (TT) (CHILDS; BRODEUR; KOCSIS, 2008) (COULOM, 2007b). Tal estrutura de dados, auxilia na avaliação de movimentos já simulados anteriormente, ganhando mais tempo para o algoritmo. Mais exemplos da utilização da TT em Go podem ser vistos em (JUNIOR; JULIA, 2014).

Diferentemente do agente Fuego, No CrazyStone toda a política de simulação é baseada no critério BT (HUNTER, 2004). Com o objetivo de fornecer uma boa distribuição probabilística de jogadas em cada episódio, o CrazyStone altera dinamicamente as avaliações dos nós na árvore envolvidos no processo de Seleção daquele episódio, conforme disposto a seguir:

- Se um nó n_x foi inserido na árvore pela primeira vez, ou seja, jamais foi simulado, é realizada uma simulação *Play-Out* a partir dele, e o resultado R_i obtido será atualizado em todos os nós que compõem o caminho traçado no processo de Seleção.

- Se tal nó foi já inserido em outros pontos da árvore, portanto, simulado previamente, ele já se encontra na TT. Nesse caso, se a profundidade corrente de n_x for no máximo igual àquela que ocorreu anteriormente, expressa na TT, n_x não será simulado e seu valor estimado presente na TT será utilizado para atualizar todos os nós pertencentes ao caminho traçado no processo de Seleção. Por outro lado, se o referido teste de profundidade fracassar, n_x será simulado novamente, e o resultado proveniente de tal simulação será utilizado para alterar não somente os nós presentes no caminho traçado durante o processo de seleção, mas também o valor estimado na TT.

No trabalho de Coulom, (COULOM, 2007b) são apresentados os resultados de experimentos em que avalia-se o agente comparando o melhor movimento indicado pelo método

Bayesiano e aquele indicado por especialistas do jogo. Em tais testes o agente apresentou ter uma boa predição de movimentos a partir do meio de jogo. Outros testes, contra o agente GNU-Go, onde CrazyStone obteve uma taxa de 90.6% e 57.1% em tabuleiros 9x9 e 19x19, respectivamente, são apresentados em (COULOM, 2007a).

3.3 GNU-Go

GNU-Go (BUMP, 2003) é uma plataforma motora, desenvolvida sob a licença GNU GPL, bastante conhecida e utilizada no ramo de *Computer Go*. Atualmente, sua versão 3.8é classificada no servidor Kiseido Go Server (KGS) (KSG$_SERVER$, 2015) como 5 Kyu. O agente GNU-Go realiza a escolha de um movimento através de várias análises baseando-se em técnicas altamente supervisionadas.

O processo de escolha de movimento é iniciado com uma fase de coleta de informações extremamente importante para o entendimento do estado corrente do tabuleiro. Primeiramente, o agente realiza a identificação dos conjuntos de pedras diretamente conectadas, os chamados *Worms*. Subsequentemente, para cada *Worm* é registrado seu tamanho e número de liberdades. Após isso, o agente realiza uma análise tática relativa a cada *Worm* identificando quais podem ser capturados diretamente. Se existir algum *Worm* em risco, o agente tentará gerar movimentos de defesa para tal grupo.

Em seguida, o agente realiza a identificação dos chamados *Dragons*, que são definidos como o conjunto de *Worms* que não podem ser desconectados. Nessa fase o agente realiza uma busca minimax para identificar quais grupos de *Worms* podem ser desconectados ou conectados um ao outro. Após a detecção dos *Dragons*, é realizada uma análise de influência e força para cada um, levando em consideração os seguintes aspectos:

- Uma estimativa do número de "olhos" de cada *Dragon*, para determinar seu risco de "vida ou morte".
- Uma estimativa do impacto de cada *Dragon* em relação aos territórios adjacentes.
- Potencial de escape de um *Dragon* em caso do mesmo ser atacado.

Após essa fase, são realizadas mais algumas análises específicas para cada *Dragon*. Terminada a fase de coleta de informações, o agente dará início à fase de geração de movimentos, cujo objetivo é disponibilizar uma lista de ações candidatas àquela jogada. A geração de movimentos candidatos, por sua vez, é realizada por uma série de funções baseadas em conhecimentos específicos do jogo. Cada função chamada irá gerar uma lista de movimentos e justificativas para cada um deles presentes na lista.

Dentre as funções geradoras de movimentos, destacam-se:

- *worm_reasons*: movimentos para ataque ou defesa de um *Worm*.

- *owl_reasons*: movimentos críticos de "vida ou morte" para cada *Dragon*.
- *semeai_move_reasons*: movimentos relevantes para corridas de capturas.
- *break_in_move_reasons*: movimentos relevantes para invasão de território inimigo.
- $fuseki$: movimentos de abertura de jogos propostos em banco de dados $fuseki$.
- *shapes*: movimentos que geram "formas" importantes no jogo. Estes são obtidos através de um banco de dados de padrões utilizados pelo agente.

O agente, em sua próxima fase, irá avaliar baseando-se nas justificativas, as listas de movimentos candidatos propostos por cada função listada acima com o intuito de gerar um valor para cada ação. Uma vez avaliados, o programa escolhe aquele de maior valor.

Como pode-se perceber o agente GNU-Go é fortemente baseado em heurísticas e conhecimento supervisionado, limitando muitas das vezes seu poder estratégico em alguns jogos.

3.4 Fuego

O agente Fuego foi construído a partir de dois projetos anteriores, o Smart Game Board (KIERULF, 1990) e o Explorer (MÜLLER, 1995), em que o primeiro consiste de uma coleção de ferramentas para desenvolvimento de jogadores automáticos e o segundo é um jogador de Go desenvolvido com os recursos oferecidos pelo Smart Game Board.

Motivado pelo sucesso alcançado pelos programas jogadores de Go Crazy Stone e MoGo, que utilizam o método Monte Carlo para busca em árvore, em 2007 Enzenberger começou a desenvolver um programa que implementava tal técnica. Inicialmente chamado apenas de UCT, este programa desenvolvido por Enzengerger, Muller and Arneson, foi renomeado para Fuego e se tornou um programa de código aberto em 2008 (ENZENBERGER et al., 2010).

Uma das principais vantagens da plataforma Fuego é que mesmo sendo um dos agentes que compõem o estado da arte, ele oferece todo seu aparato de pesquisa gratuitamente e livre, possibilitando assim que qualquer pessoa possa estudar, utilizar ou até mesmo melhorar o trabalho já realizado.

A Fuego é uma plataforma de *software* e não apenas uma biblioteca com funcionalidade limitada. Um grande número de classes e funções é encontrado na plataforma que está dividida em cinco bibliotecas principais, GtpEngine, SmartGame, Go, SimplePalyers e GoUct (LIN, 2009), como ilustrado na Figura 8.

A biblioteca GtpEngine oferece uma implementação abstrata do protocolo *Go Text Protocol*, o qual foi utilizado pela primeira vez no programa GNU Go e ganhou ampla

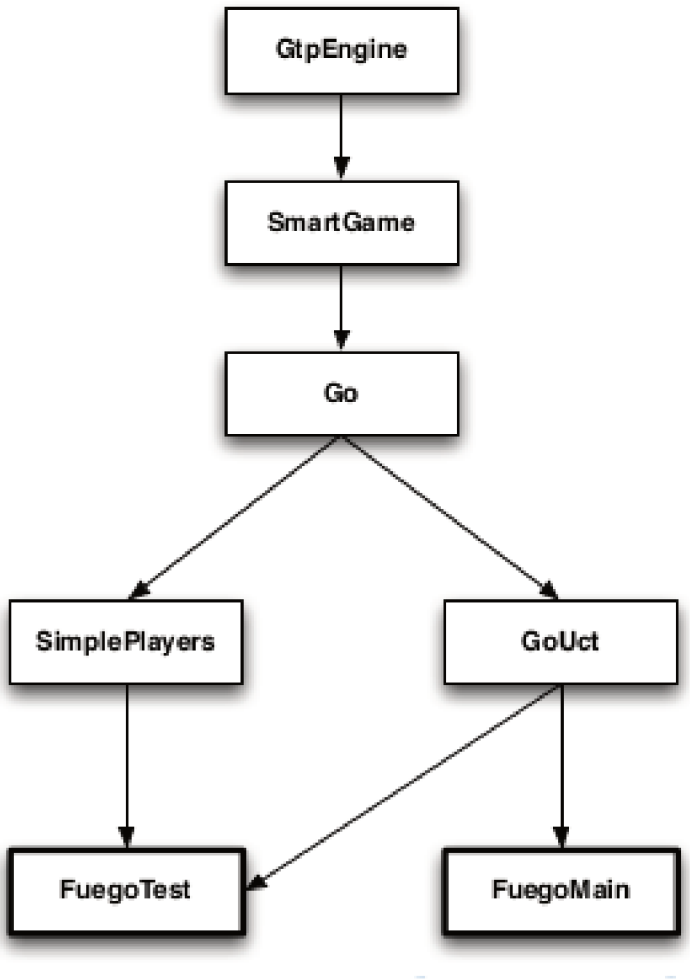

Figura 8 – Arquitetura em primeiro nível da plataforma Fuego.

aceitação na comunidade desenvolvedora de agentes para o jogo de Go. O protocolo foi, posteriormente, adaptado para outros jogos como o Amazons, Othello, Havannah e Hex. Basicamente, tal biblioteca recebe e interpreta os comandos externos, invocando as rotinas necessárias e específicas de cada um.

A biblioteca SmartGame contém funcionalidades úteis a jogos em geral, como classes e funções que ajudam a representar o estado do tabuleiro de um jogo. As classes mais complexas e sofisticadas em tal biblioteca são as implementações da busca Alfa-Beta e do algoritmo MCTS. Ambas compõem classes separadas podendo serem utilizadas de forma independente.

A biblioteca Go é implementada a partir da SmartGame e oferece funcionalidades específicas do jogo de Go. A classe mais importante se trata de uma implementação bastante eficiente do tabuleiro de Go. Ela permite armazenar o histórico do jogo, sendo possível desfazer movimentos já realizados, o que é muito importante para agentes que trabalham com simulações de movimentos. Além do mais, ela mantém informações relativas ao número de liberdades das peças e grupos presentes no tabuleiro assim como verifica a legalidade de cada movimento realizado. Em tal biblioteca é possível configurar

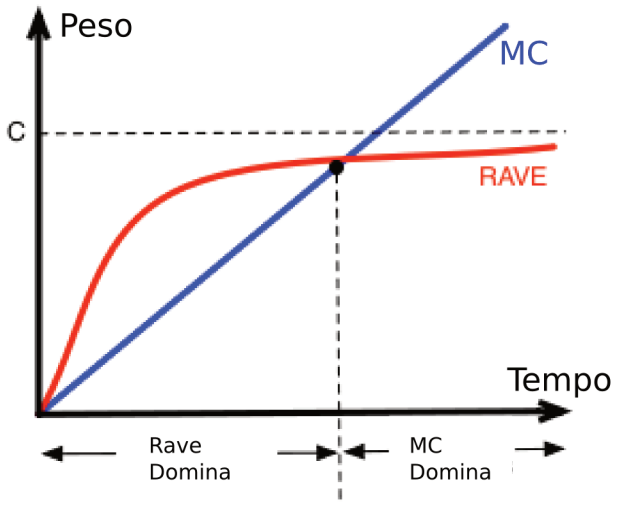

Figura 9 – Relação entre os pesos utilizados entre as estimativas RAVE e MC de acordo com o tempo.

diversas convenções de pontuação no final do jogo.

SimplePlayers, assim como o nome sugere, reúne algoritmos simples de jogadores automáticos para o jogo, como por exemplo, um jogador que simula movimentos escolhidos de acordo com uma política de simulação aleatória e outro que procura maximizar uma função de influência em tal política. Tais jogadores são, na maioria dos casos, utilizados para testes e referências.

GoUCT e FuegoMain correspondem às bibliotecas que implementam a parte principal do motor Fuego. Fuegotest é uma interface de testes para as funcionalidades presentes nas bibliotecas SmartGame e Go.

A plataforma Fuego conta também com um jogador de Go com o mesmo nome. Durante seu processo de Seleção o agente utiliza uma política de árvore baseado nos métodos de simulação Monte Carlo e RAVE apresentados no capítulo 2.5. A Figura 9 apresenta a relação dos pesos relacionados ao RAVE e estimativa Monte Carlo, assim como mostrado na equação 6, de acordo com o tempo.

Uma vez terminado o processo de seleção o algoritmo parte para a expansão da árvore de busca. Em cada episódio executado, antes da inserção de um movimento na árvore de busca o agente executa uma rotina, chamada *Prior-Knowledge* de inicialização dos valores de um nó. Tal rotina é baseada em heurísticas retiradas de experimentos com outros agentes, e tem o propósito de acelerar o processo de estimativa de um nó. Uma análise mais diligente é apresentada no capítulo 4 seção 4.2.

Após o processo de Expansão da árvore de busca, é iniciado o processo de simulação *play-out*. Um conjunto de diretrizes, chamado de política de simulação é estabelecido, então, para a execução dos movimentos durante esse processo. Tal política é composta por algumas verificações de conhecimento específico do jogo com o intuito de melhorar

a qualidade das simulações melhorando assim o poder geral de tomada de decisões do agente. Uma análise da limitação de políticas aleatórias é melhor descrita no capítulo 4 seção 4.4.

Além do mais, o agente conta com um livro de abertura de movimentos para as ações iniciais no jogo. Isso contribui bastante para seu bom desempenho.

A plataforma Fuego é utilizada por diversos jogadores automáticos, destacando-se entre eles o programa MoHex (ARNESON; HAYWARD; HENDERSON, 2008), desenvolvido para o jogo Hex e de nível profissional, o Tsumego Explorer (KISHIMOTO, 2005), um dos melhores programas solucionadores de problemas do tipo "vida e morte" no jogo de Go, e o agente RLGO (SILVER; SUTTON; MÜLLER, 2007), o qual aprende formas locais por diferenças temporais com o mínimo de conhecimento específico de Go.

Capítulo 4

Go-Ahead

Este capítulo apresenta o Go-Ahead, um agente inteligente para o jogo de Go que tem como objetivo introduzir técnicas de aperfeiçoamento das decisões a serem tomadas pelo jogador através da otimização e especialização do processo de pré-avaliação dos movimentos candidatos a serem inseridos na árvore de busca Monte-Carlo durante sua fase de expansão. O refinamento dessa pré-avaliação é obtido através de informações dinâmicas obtidas nos *Play-Outs*. Para alcançar essa meta, o agente estende a arquitetura do Fuego com a inclusão do módulo Avaliação Dinâmica de Movimentos (ADM), que por sua vez é composto por três submódulos principais:

1. Submódulo de Atualização dos Dados: este módulo é o responsável por receber todos os dados das simulações executadas até o momento. Uma vez recebidos, estes dados serão filtrados e agrupados aos nós já presentes na memória. A cada vez que uma etapa de *Play-Out* é concluída no módulo MCTS, são realizadas inúmeras atualizações dos dados presentes na memória desse módulo. Esse módulo mantém todos os dados guardados na memória RAM com o intuito de acelerar o trabalho do módulo descrito a seguir.

2. Submódulo Banco de Dados: módulo responsável por realizar uma busca nos dados relativos às simulações previamente realizadas presentes na memória RAM do computador. Este módulo recebe como entrada uma lista de movimentos candidatos a serem inseridos na árvore de busca Monte Carlo, realiza uma busca em sua memória interna, e retorna uma lista com os movimentos recebidos como entrada e seus respectivos valores estimados.

3. Submódulo de Avaliação Final: finalmente, este módulo tem a responsabilidade de combinar os dados recebidos através do módulo de *prior-knowledge* do agente Fuego com os dados estimados dinamicamente neste processo recorrente. Essa combinação é dada através de um balanço entre os dados obtidos com heurísticas e aqueles obtidos ao final de cada *Play-Out*.

Uma das propostas do agente é utilizar as informações dinâmicas providas durante as simulações dos episódios para atenuar o caráter supervisionado da busca MCTS realizada pelo jogador Fuego. Tal supervisão se refere ao uso da estratégia de *prior knowledge* (Conhecimento prévio). Para esse fim, o agente utiliza-se de todos os dados gerados durante a fase de *play-out*, de forma a extrair conhecimento específico do jogo corrente e combiná-lo com aquele já obtido via heurísticas de *Prior-Knowledge* codificadas explicitamente no agente Fuego. De fato, a abordagem aqui apresentada assemelha-se à técnica AMAF aplicada aos termos da heurística *Prior Knowledge* conforme discutido em maiores detalhes na subsecção 4.4.2.

4.1 Arquitetura Geral

O agente Go-Ahead foi desenvolvido sobre o conhecido agente Fuego e melhora o processo de busca através da utilização de técnicas estatísticas aplicadas à vasta massa de dados gerados na fase de *play-out* do processo de busca MCTS. Analogamente ao jogador Fuego, Go-Ahead também utiliza o algoritmo de busca MCTS descrito em 2.3. Dessa forma, o mecanismo de inteligência do agente consiste, principalmente, na construção iterativa de uma árvore de busca Monte Carlo, utilizando uma combinação dos algoritmos RAVE e Monte Carlo como política da fase de seleção..

Entretanto, diferentemente do jogador Fuego, Go-Ahead modifica o processo de pré-avaliação dos nós candidatos a serem inseridos na árvore de busca Monte Carlo durante a fase de expansão. Essa modificação se dá através da comunicação do módulo *prior knowledge* com o módulo ADM, que por sua vez é composto por três submódulos, como ilustrado na Figura 10.

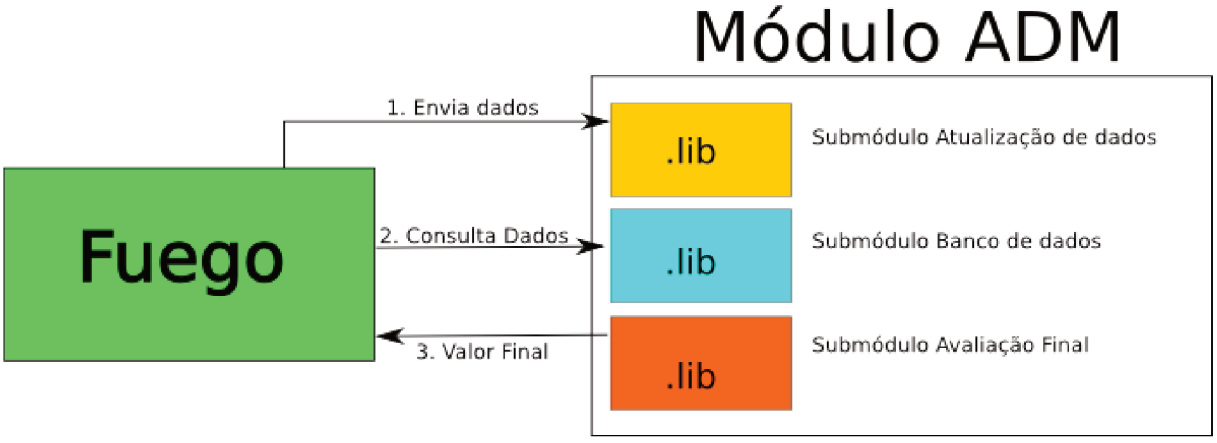

Figura 10 – Arquitetura geral do agente Go-Ahead

Na arquitetura apresentada na Figura 10, cada submódulo foi codificado como uma biblioteca independente, compartilhando apenas a uniformidade das estruturas de dados

para uma posterior comunicação entre os submódulos. A arquitetura baseada em bibliotecas independentes atribui ao agente uma característica de baixo acoplamento e alta coesão entre as entidades do módulo ADM, possibilitando assim modificações nas políticas de atualização, consulta ou estimativa final dos dados, sem um maior impacto no agente como um todo.

Como descrito anteriormente, durante o processo de busca Monte Carlo, são executados inúmeros episódios de busca, cada episódio compondo-se das quatro fases principais do algoritmo MCTS: Seleção, Expansão, *Play-out* e Retro propagação.

Naturalmente, o processo de seleção culmina com a formação de um caminho traçado a partir da raiz até um nó folha da árvore de busca. Após a geração desse caminho, inicia-se a fase de expansão da árvore, que é onde de fato a árvore de busca irá crescer, atribuindo ao agente uma visão em profundidade melhor, o que o possibilitará, por consequência, ler mais jogadas à frente de seu oponente, característica fundamental para jogadores de jogos de tabuleiros.

Previamente à fase de expansão, o algoritmo em suas versões mais modernas, passa pela fase de pré-avaliação de um nó que será responsável por iniciar um nó com valores relevantes ao processo de busca. No agente Fuego isso é realizado através do módulo *prior knowledge*, o qual atribui, caso existam, valores relativos à qualidade e ao número de execuções de um movimento em buscas Monte Carlo. Esse valores são codificados explicitamente no código-fonte do agente, limitando o conhecimento relativo aos nós candidatos a serem inseridos na árvore de busca àquele definido pela heurística *Prior-Knowledge*, fato que impossibilita o agente perceber certas nuances específicas ao jogo atual.

No agente Go-Ahead, esse processo é feito de uma forma sucintamente diferente. Ao acionar o módulo *Prior Knowledge*, o agente realiza todos os processos de pré-avaliação dos movimentos candidatos de acordo com as regras embutidas manualmente em código-fonte, assim como o agente original. Uma vez pré-avaliados, os movimentos candidatos são repassados, em uma lista, ao módulo ADM para refinamento da avaliação ocorrida previamente. Assim que recebe tal lista para um refinamento de avaliação, o módulo ADM repassa os dados para seu submódulo de banco de dados. Esse último, por sua vez, é responsável por buscar, em uma estrutura de dados mantida na memória *heap* do programa, cada movimento contido na lista, recuperando seus valores estimados até o presente momento.

Assim que todos os dados são recuperados na memória eles são repassados, juntamente com os dados obtidos no módulo *Prior Knowledge*, para o submódulo de Avaliação Final do módulo ADM. Esse submódulo é responsável por combinar linearmente os dados obtidos até o momento através de uma expressão apresentada posteriormente neste capítulo. Essa combinação visa elevar a acuidade da estimativa do valor dos movimentos através da utilização de conhecimento provindo das heurísticas - comprovadamente eficazes - e das estimativas dinâmicas baseadas nos *Play-Outs* efetuadas no módulo ADM.

Finalmente, após realizada tal combinação, a lista de movimentos contendo os movimentos associados a suas respectivas pré-avaliações com acuidade aprimorada é repassada de volta ao módulo *Prior-Knowledge*. O módulo *Prior Knowledge*, por sua vez, repassa os dados ao módulo MCTS para que a busca continue naturalmente. Posteriormente, no mesmo episódio, durante a fase de retro propagação, o módulo MCTS atualiza os valores dos nós da árvore de busca. Além disso, ele repassa os daos resultantes da simulação *Play-Out* ao módulo ADM. Este último, por sua vez, transfere tais dados a seu sub-módulo de Atualização de Dados a fim de que ele os utilize durante o cumprimento de uma das seguintes tarefas: inserir na estrutura heap do ADM aqueles dados que se refiram a estados aainda não presentes em tal estrutura; ou atualizar as informações dos nós dessa estrutura que sejam coincidentes com os estados relativos a tais dados.

As próximas subseções apresentam, mais detalhadamente, os módulos que compõem a arquitetura geral do Go-Ahead.

4.2 Análise do módulo de *Prior-Knowledge*

Conforme descrito anteriormente, o módulo de *Prior-Knowledge* é responsável por atribuir valores aos movimentos candidatos a serem inseridos na árvore de busca, mais especificamente, ele inicializa os dados de *valor* e *contador* de um determinado movimento. Dessa forma, tal módulo funciona como um catalisador da busca, atribuindo valores iniciais aos movimentos candidatos guiado por heurísticas. Os valores e regras definidos pelas heurísticas desse módulo foram obtidos por experimentos otimizados de auto jogo, jogos contra o agente MoGo v.3 e contra o jogador automático GNU Go 3.6 em tabuleiros 9x9 e 19x19. Na prática, os valores utilizados para tabuleiros 9x9 são também utilizados para todos os tabuleiros de tamanho menor que 15, e os valores configurados para 19x19 utilizados nos tabuleiros de tamanhos 15 a 19.

No módulo, os movimentos são inicializados na árvore de busca, sempre com valores positivos, para o jogador que possui a vez.

Bônus adicionais são dados a movimentos que pertencem aos padrões de tamanho 3x3, utilizados também pela política de *Play-Out* a saber: movimentos que colocam o oponente em posição de atari e todos os movimentos que se encontram a uma certa distância do último movimento jogado. Inversamente, movimentos que caracterizam *auto atari* são penalizados. Sendo assim, os valores utilizados na inicialização dependem da ocorrência de tais movimentos na classe de *Prior-Knowledge* assim como no tamanho do tabuleiro utilizado.

Os pseudo códigos mostrado em 2 e 3, ilustram a implementação e utilização dos algoritmos de conhecimento prévio.

Algoritmo 2 Algoritmo de Expansão: Fuego

Entrada: Árvore MCTS: T

Saída: Árvore MCTS expandida: T

```
1  início
2      Lista<Movimentos> moves;
3      GeraTodosOsMovimentosLegais(moves);
4      ProcessaPosição(moves);  %comment : HeurísticaPrior − Knowledge %
5      InsereFilhosNaÁrvore(T, moves);
6      SelecionaMelhorFilho(moves);
7  fim
```

Algoritmo 3 ProcessaPosição: Fuego

Entrada: Lista<Movimentos> *moves* (não avaliada)

Saída: Lista<Movimentos> *moves* (avaliada)

```
1  início
2      bool tabuleiro_pequeno = (TamanhoTabuleiro(Tab) < 15);
3      para p ← PrimeiraPosiçãoTabuleiro(Tab) até ÚltimaPosiçãoTabuleiro(Tab) faça
4          se p_i ! = vazio então
5              continue;
6          fim
7          se SelfAtari(p_i) então
8              Inicializa(p_i, 0.1f, tabuleiro_pequeno? 9 : 18);
9              senão se Atari(p_i) então
10                 Inicializa(p_i, 0.8f, tabuleiro_pequeno? 9 : 18);
11                 senão se Padrão(p_i) então
12                     Inicializa(p_i, 0.6f, tabuleiro_pequeno? 9 : 18);
13                     senão
14                         Inicializa(p_i, 0.4f, tabuleiro_pequeno? 9 : 18);
15                     fim
16                 fim
17             fim
18         fim
19     fim
20     TransfereValores( Tab, moves);
21 fim
```

Como mostrado no pseudo código representado pelo Algoritmo 2, por questões de otimização algorítmica e estrutural, o agente insere vários nós filhos de uma só vez na árvore de busca (linha 5), diferentemente do algoritmo MCTS em sua forma original, o

qual efetua a inserção de um nó filho por iteração. Com isso, o Go-Ahead consegue ganhar performance e utilizar memória contígua na prática, o que em termos computacionais otimizará o acesso aos dados através de uma melhor utilização da memória cache.

O Algoritmo 2 representa o pseudo código para o processo de expansão da árvore de busca Monte Carlo. Ele recebe como entrada uma referência a uma árvore de busca Monte Carlo T, e retorna a mesma referência já com os nós expandidos. No algoritmo, primeiramente é declarada uma lista de movimentos chamada *moves* (linha 2). Logo a seguir, todos os movimentos legais são gerados pela função *GeraTodosOsMovimentos* de acordo com o tabuleiro T e guardados na lista *moves* (linha 3). Uma vez que todos os movimentos legais estejam disponíveis, é hora de atribuir um valor aos mesmos de forma a otimizar o processo de busca subsequente. Essa inicialização otimizada dos movimentos disponíveis é realizada através da função *ProcessaPosição* (linha 4). Assim que é finalizada tal inicialização dos movimentos, dá-se início ao processo de inserção dos filhos na árvore de busca. Como mencionado anteriormente, todos os filhos são inseridos em uma única iteração por uma questão de otimização de tempo e espaço computacional. Após o término do processo de inserção de todos os nós filhos na árvore de busca, o algoritmo seleciona aquele com melhor avaliação (linha 6). A partir de tal nó selecionado se dará o início de todo o processo de simulação *Play-Out*.

O Algoritmo 3 representa o pseudo código para o processo de inicialização dos valores dos movimentos legais disponíveis de acordo com o tabuleiro Tab. O algoritmo recebe como entrada uma lista de movimentos (movimentos legais disponíveis gerados pela função $GeraTodosOsMovimentosLegais$) ainda não avaliados, retornando a mesma lista com cada um de seus movimentos avaliados de acordo com algumas regras. Primeiramente, o algoritmo verifica qual é o tipo do tabuleiro (linha 2). Aqui, são considerados apenas dois tipos: tabuleiro pequeno (menores que 15x15) e tabuleiros normais (maior ou igual a 15x15). Em seguida, o algoritmo executa um laço, passando por cada posição do tabuleiro (linha 3). Se tal posição já está ocupada, ele apenas continua o processo. Caso contrário, ele checa tal posição de acordo com as seguintes regras:

1. Self-Atari (linha 7): A posição p indica self-atari? Caso afirmativo, ela recebe um valor baixo, no caso, $0.1f$ como valor de qualidade e 9 ou 18 como valor do contador, de acordo com o tamanho do tabuleiro.

2. A posição p indica atari positivo (linha 9)? Caso afirmativo, ela recebe $0.8f$ como valor de qualidade e 9 ou 18 como valor de contador, de acordo com o tamanho do tabuleiro.

3. A position p faz parte de algum padrão de jogada conhecida (linha 11)? Caso afirmativo, recebe $0.6f$ como valor de qualidade e 9 ou 18 como valor de contador, de acordo com o tamanho do tabuleiro.

4. Finalmente, se a posição p não se encaixa em nenhuma das alternativas anteriores, é-lhe atribuido $0.4f$ como valor de qualidade e 9 ou 18 como valor de contador, de acordo com o tamanho do tabuleiro (linha 14).

Após o processamento e avaliação de todas as posições vazias do tabuleiro, são realizadas as transferências de valores para os movimentos contidos em *moves* (linha 20).

4.2.1 Limitações do módulo *Prior-Knowledge*

Como mencionado previamente, os valores utilizados no módulo foram obtidos de forma empírica com o auxílio de outros agentes. Sendo assim o módulo se torna de certo modo engessado, uma vez que os parâmetros, regras e valores são codificados de forma explícita no código fonte do agente Fuego.

Consequentemente, a avaliação dos movimentos se dá de forma genérica, atribuindo na maioria das vezes os mesmos valores a movimentos completamente diferentes no escopo do jogo corrente. Isso, de certa forma, prejudica a performance do agente, uma vez que o mesmo não é capaz de captar as nuances específicas de cada movimento no jogo atual.

A Figura 11 elucida, de forma simplificada, a limitação do módulo de conhecimento prévio. Como ilustrado na Figura 11, os movimentos A, B, C e D possuem a mesma avaliação, apesar de causarem um impacto completamente diferente no jogo. Embora estejam muito proximamente localizados, tais movimentos deveriam ter valores diferentes.

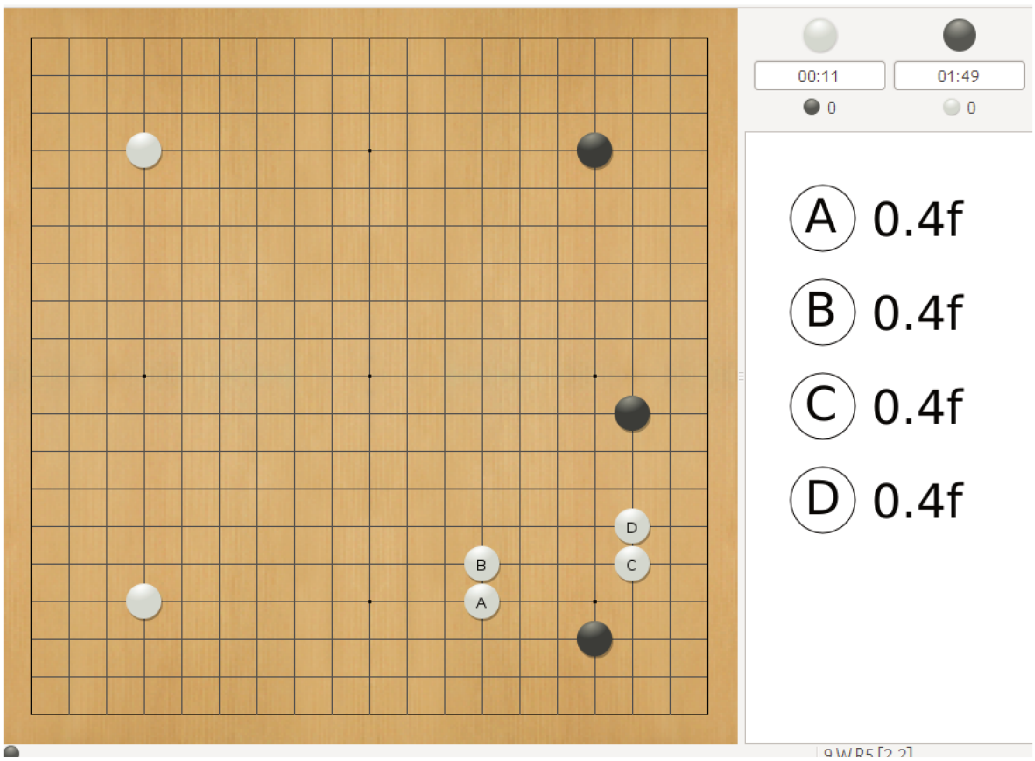

Figura 11 – Limitações do módulo de conhecimento prévio

Na Figura 11, o movimento A é o mais indicado, de acordo com especialistas e profissionais no jogo. A formação realizada pelas peças pretas no tabuleiro de tal figura representa uma abertura de jogo conhecida como abertura Chinesa. É um padrão bastante forte e pode colocar o jogador preto em clara vantagem caso o jogador branco não saiba neutralizar a influência exercida por tal *framework*. Ao mesmo tempo a jogada B não é considerada tão boa quanto A, uma vez que se encontra um pouco mais distante, causando um impacto menor na influência do padrão Chinês de abertura. Finalmente, as jogadas C e D, apesar de serem consideradas aproximações comuns à peça preta localizada na parte mais inferior do tabuleiro, não são respostas boas ao *framework* chinês de abertura.

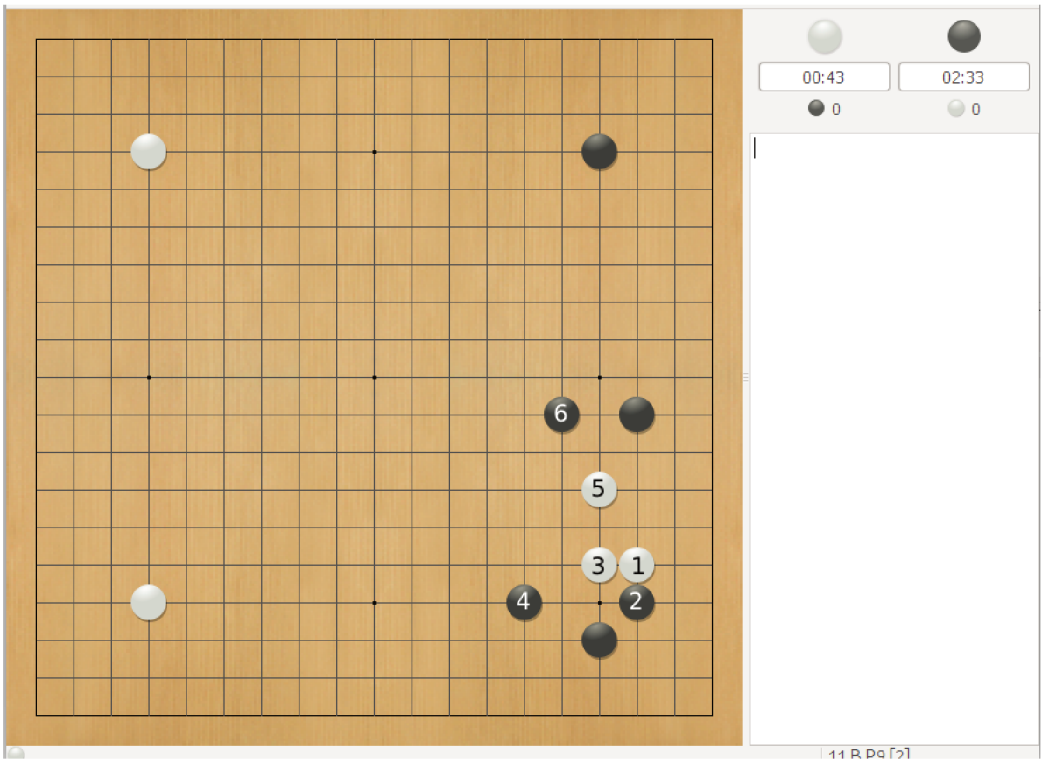

Figura 12 – Possível sequência para a jogada C representada na Figura 11.

Por exemplo, a Figura 12 representa uma possível sequência de jogadas em resposta à jogada C da Figura 11. Tal jogada é considerada muito funda no *framework* chinês montado pelas peças pretas. Isso, normalmente, levaria à sequencia de movimentos demonstrados pela Figura 12, o que, claramente, deixaria as peças pretas em uma posição bem mais confortável do que seu oponente.

Tal limitação possibilitou o levantamento da seguinte questão: é possível aumentar a acuidade das informações do módulo *Prior-Knowledge* e, simultaneamente atenuar seu caráter supervisionado?

4.2.2 Uma alternativa

Durante a fase de *play-out* do algoritmo MCTS, milhares de simulações, senão milhões, são realizadas e posteriormente descartadas. Tais simulações servem como parâmetro para a avaliação de um caminho traçado na árvore de busca durante a fase de seleção.

Sendo assim, o agente acaba desperdiçando bastante conhecimento gerado de forma online e, principalmente, específico ao jogo atual. Nessa direção, o trabalho aqui apresentado propõe a utilização de tais dados em combinação com a força já comprovada das heurísticas de conhecimento prévio para a a criação de um agente com um poder melhor de pré-avaliação durante a fase de expansão.

Uma das principais razões em se utilizar os dados gerados durante a fase de *play-out* é o fato de que tal conhecimento modela muito bem a situação do jogo corrente. Sendo assim, um movimento muito bem avaliado pelo módulo de conhecimento prévio pode não ser a melhor opção para o jogo específico em questão. Desse modo o agente Go-Ahead modifica o algoritmo original não descartando o conhecimento gerado na fase de *play-out*, mas sim acumulando-o e aplicando técnicas para sua devida utilização posteriormente.

Por exemplo, para comprovar a pertinência da investigação da modificação proposta neste trabalho, foi criado um cenário baseado na situação apresentada pela Figura 11. Em tal cenário limitamos o número de respostas das peças brancas às quatro jogadas A, B, C, D apresentadas na figura. Posteriormente, foram executadas 200.000 simulações *play-out* (realizadas pelo agente Fuego) para cada uma das jogadas.

Como apresentado na Figura 13, após 200.000 simulações partindo de cada uma das jogadas indicadas, o agente apresentou uma taxa de vitória de aproximadamente 53% para a jogada A, 50% para a jogada B, 44% para a jogada C e 47% para a jogada D. Isso claramente indica que a jogada A é levemente superior em relação às outras 3 jogadas propostas. Vale ressaltar que em um cenário real, o agente considera todas as outras posições vazias no tabuleiro, o que diminui o número de simulações por nó, podendo afetar assim a confiabilidade dos valores obtidos. Por causa dessa limitação a proposta é utilizar o conhecimento gerado através das simulações *play-out* em conjunto com o módulo de heurística já existente.

Logo, a construção do agente Go-Ahead visa combinar a força confirmada pelas heurísticas presentes no módulo de conhecimento prévio com o dinamismo e adaptabilidade das estimativas geradas pelo módulo ADM.

4.3 Módulo de Avaliação Dinâmica de Movimentos (ADM)

O módulo de avaliação dinâmica de movimentos (ADM) é responsável por toda a ação com relação às modificações realizadas no agente Fuego, gerando, assim, o agente

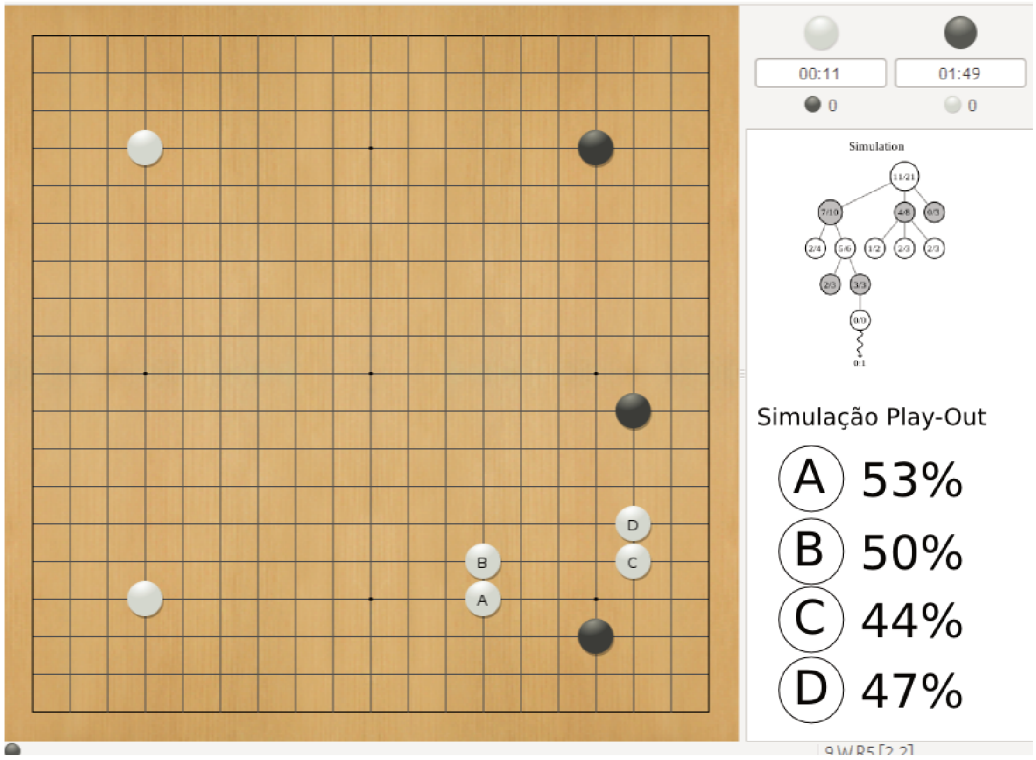

Figura 13 – Resultado após 200.000 simulações executadas (de acordo com o agente Fuego) para cada uma das 4 jogadas apresentadas no cenário acima.

Go-Ahead.

De um modo geral ele modifica a busca MCTS original realizando um balanço entre os valores obtidos através do módulo de conhecimento prévio e aqueles gerados dinamicamente durante as fases de *Play-Out* executadas nas buscas efetuadas pelo agente.

A Figura 14 ilustra esse processo de uma forma geral.

1. Após a fase de seleção, todos os nós candidatos (nós filhos do último nó escolhido no processo de seleção) são avaliados com o valor pré estimado pelo módulo ADM;

2. Todos os nós filhos são adicionados à árvore, por questões de performance computacional, mas só aquele com o maior valor pré estimado será simulado em tal episódio;

3. A partir dele é disparada a fase de *play-out*;

4. Assim que a simulação *play-out* é concluída, o conjunto de movimentos simulados, bem como o reforço obtido (vitória ou derrota), diferentemente do Fuego, são repassados ao módulo ADM para que o mesmo possa atualizar seus dados em memória;

5. Dá-se início à fase de retro propagação, de forma a atualizar os nós presentes na árvore de busca.

6. Todo o processo é disparado novamente (limitado ao número de episódios que foi previamente estabelecido no processo de busca).

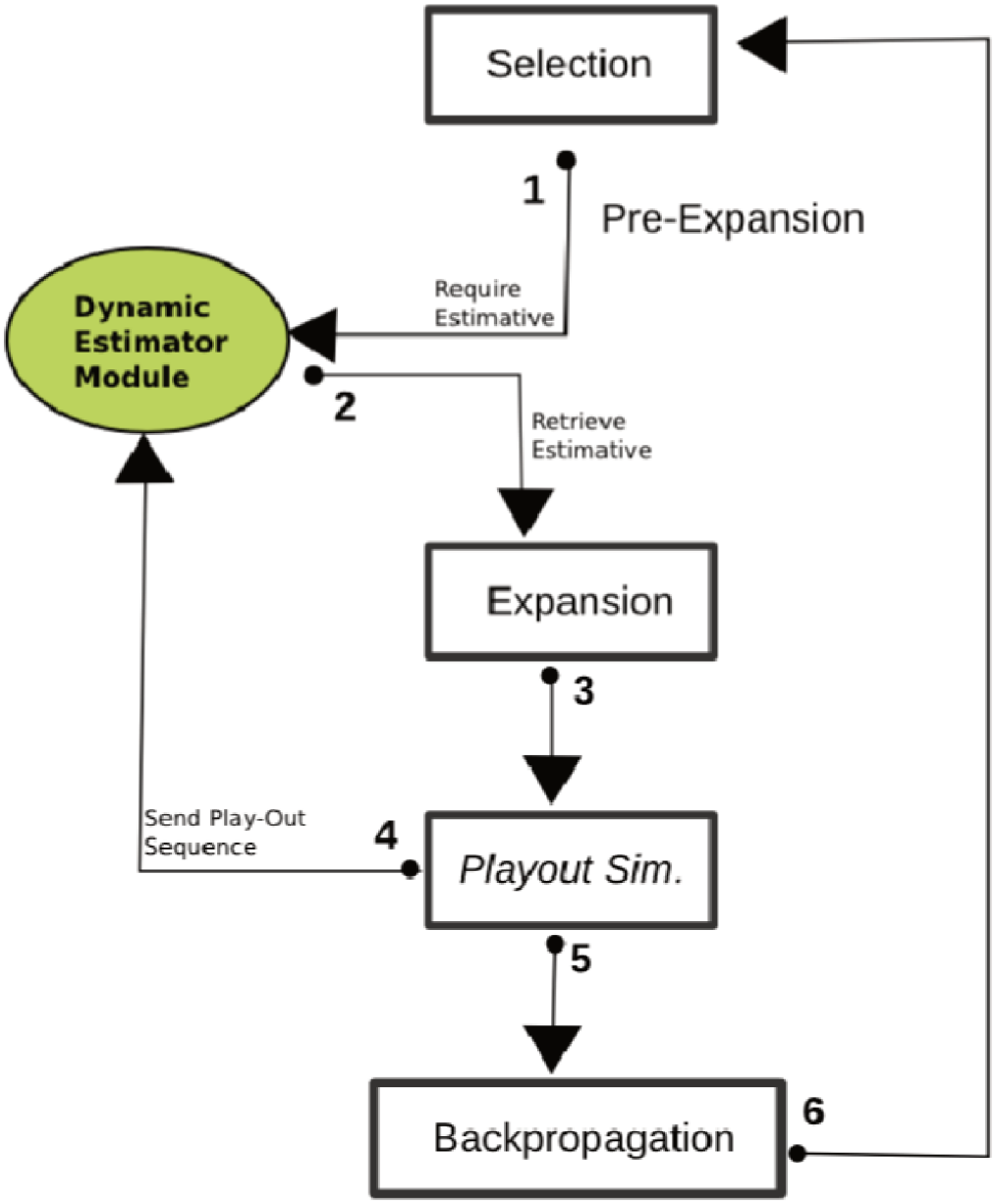

Figura 14 – Esquema Geral da arquitetura do agente Go-Ahead.

O módulo ADM, por sua vez, é dividido nos 3 submódulos principais descritos a seguir.

4.3.1 Submódulo Banco de Dados ADM

O submódulo banco de dados é responsável por implementar e manter todas as estruturas de dados que armazenarão os dados da fase de *play-out* recebidos através do módulo

MCTS.

Sua principal estrutura de dados é uma tabela *hash*, a qual armazena as informações úteis dos movimentos simulados durante a fase de *play-out*. Uma tabela *hash* é uma estrutura de dados especial que associa chave a valor. Ela tem como objetivo principal receber como entrada uma chave simples, realizar uma busca rápida e retornar o valor desejado. O uso de uma tabela *hash* permite o acesso rápido aos dados com complexidade média de O(1) por operação, alcançando O(N) no pior caso (onde N é o número de interseções no tabuleiro) (SZWARCFITER; MARKENZON, 1994). O pior caso refere-se a situações nas quais não será possível concluir a inserção de dados devido a ocorrências de colisões em todas as posições possíveis.

O submódulo conta com uma política de acesso aos dados que pode ser facilmente configurada externamente. Ele foi desenvolvido levando em consideração os conceitos de baixo acoplamento e alta coesão da engenharia de software. Sendo assim, apesar de os submódulos serem reutilizáveis, eles podem também ser substituídos por outros que utilizem políticas de acesso aos dados diferentes daquelas já utilizadas.

É importante salientar que a utilização de um *array* com alocação contígua de memória seria suficiente para atender os requisitos funcionais do submódulo. No entanto, visando algumas modificações e implementações futuras, optou-se pela utilização de uma tabela *hash*.

O objetivo da tabela *hash* é acelerar o armazenamento e o acesso aos dados relativos aos nós já simulados no módulo MCTS. Cada nó dentro do módulo ADM é composto pelas três variáveis, representadas na estrutura abaixo:

```
struct {
unsigned int move_number;
unsigned int move_counter;
double move_value;
} node_struct;
```

onde: *move_number* é uma variável do tipo *inteiro* representando a chave do próprio movimento; *move_counter* é outra variável do tipo *inteiro* a qual indica quantas vezes tal movimento foi simulado no jogo corrente; finalmente, *move_value* é uma variável do tipo *double* representando a taxa de vitórias acumuladas nos *play-outs* em que tal movimento apareceu nas sequências de simulação.

Desse modo, toda vez que um registro deve ser atualizado ou consultado, ele é passado ao módulo de banco de dados que aplicará uma função *hash* sobre seu atributo *move_number* para poder encontrar sua posição exata na memória. Esse processo é ilustrado através da Figura 15.

Durante o processo de atualização, caso um determinado movimento não possua nenhuma registro relacionado, o módulo cria um novo registro na tabela e adiciona tal registro com os valores atualizados. Opostamente, durante o processo de consulta, caso

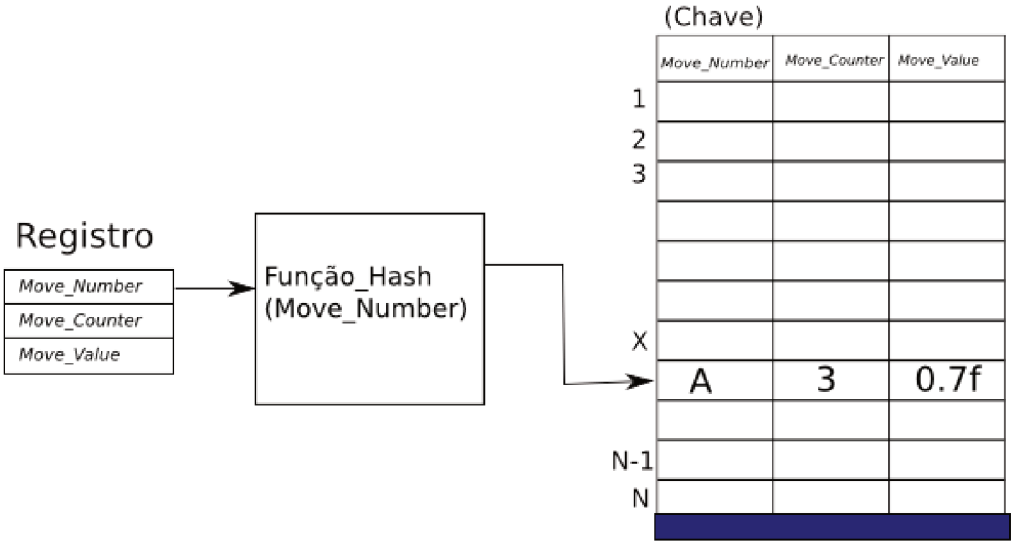

Figura 15 – Estrutura interna do módulo de banco de dados.

um determinado movimento não possua nenhum registro relacionado, o módulo retorna um valor *Nulo*, indicando que não houve nenhum registro de tal movimento.

4.3.2 Submódulo de Atualização de Dados ADM

O Submódulo de atualização de dados é responsável por implementar as políticas de atualização dos dados mantidos pelo submódulo Banco de Dados. Uma vez que uma sequência *play-out* é finalizada, toda a sequência de movimentos, juntamente com a recompensa obtida, é repassada ao módulo ADM, que por sua vez itera pela mesma realizando, para cada movimento, uma consulta e uma atualização/inserção de registro na tabela *hash*.

O algoritmo a seguir elucida bem esse processo de atualização.

Algoritmo 4 AtualizaRegistros

Entrada: Lista<*node_struct*> *sequence*; Double R

Saída: Dados atualizados de acordo com a política de atualização.

```
1  início
2      node_struct* rx;
3      para reg ← PrimeiraPosição(sequence) até ÚltimaPosição(sequence) faça
4          rx ← DataBaseModule.Get(reg_i.move_number);
5          se rx != NULL então
6              rx.move_counter ← rx.move_counter + update_jump;
7              rx.move_value ← rx.move_value + α × [R − (β × rx.move_value)];
8          fim
9          senão
10             DataBaseModule.Insert(reg_i);
11         fim
12     fim
13 fim
```

Como pode ser notado no algoritmo 4, assim que uma sequência é recebida, o módulo de atualização de dados itera sobre a mesma tomando um movimento(registro) por vez (linha 3). Para cada registro reg_i ele procura um correspondente na memória através do objeto $DataBaseModule$, o qual representa uma instância do módulo de Banco de Dados explicado em 4.3.1 (linha 4). Caso uma correspondência para tal registro seja encontrada, ele a atualiza de acordo com as regras de atualização (linhas 6 e 7):

$$C_{(X)} \leftarrow C_{(X)} + update\_jump \tag{7}$$

$$V_{(X)} \leftarrow V_{(X)} + \alpha \times [R - (\beta \times V_{(X)})] \tag{8}$$

A regra de atualização 8 foi inspirada no método das diferenças temporais (TESAURO, 1992), com algumas sutis modificações. Nela $V_{(X)}$ representa o atributo *move_value* do registro X, α representa uma taxa de aprendizado, R representa a recompensa obtida ao final de uma simulação *play-out* e β representa a utilização (ou não) do valor anterior na atualização.

É muito importante ressaltar que no escopo desse trabalho as constantes α e β foram mantidas sempre com os valores 1 e 0, respectivamente, uma vez que o objetivo maior desse trabalho é avaliar o impacto do conhecimento gerado nas simulações *play-out* em conjunto com as heurísticas *PriorKnowledge*.

No entanto, de acordo com objetivos e planejamentos futuros, o módulo foi preparado para suportar uma política de atualização através do método das diferenças temporais em que, ao invés de repassar ao agente um único reforço após concluído um *Play-Out*

(ou seja, a simulação completa de um jogo), serão repassados reforços intermediários estimados após cada simulação de movimento de um episódio.

Na regra de atualização 7 $C_{(X)}$ representa o atributo *move_counter* do registro X e *update_jump* representa uma constante de atualização, simbolizando o salto que tal contador deve realizar. Tal constante é muito útil quando se quer representar mais de uma simulação por atualização de registro. No entanto, para o presente trabalho ela foi mantida com o valor 1, pois realizamos apenas uma atualização por simulação *play-out*.

Por outro lado, quando uma correspondência para o registro em questão não é encontrada, é realizada uma inserção desse registro na memória, assim como demonstrado na linha 10 do algoritmo 4.

4.3.3 Submódulo de Avaliação Final ADM

O submódulo de avaliação final é o responsável por realizar todo o balanço entre o conhecimento adquirido pelo módulo ADM e aquele representando pelas heurísticas *Prior-Knowledge*.

Durante a fase de expansão, o agente Go-Ahead, consulta o módulo ADM para poder obter a avaliação de todos os movimentos legais gerados.

Algoritmo 5 Algoritmo de Expansão: Go-Ahead

Entrada: Árvore MCTS: T

Saída: Árvore MCTS expandida: T

```
1 início
2     Lista<Movimentos> moves;
3     GeraTodosOsMovimentosLegais(moves);
4     ModuloADM.ProcessaPosição(moves); %comment: Avaliação realizada pelo módulo
      ADM %
5     InsereFilhosNaÁrvore(T, moves);
6     SelecionaMelhorFilho(moves);
7 fim
```

Algoritmo 6 ProcessaPosição: Modulo ADM Go-Ahead

Entrada: Lista<Movimentos> *moves* (não avaliada)

Saída: Lista<Movimentos> *moves* (avaliada)

```
1  início
2      bool tabuleiro_pequeno = (TamanhoTabuleiro(Tab) < 15);
3      para p ← PrimeiraPosiçãoTabuleiro(Tab) até ÚltimaPosiçãoTabuleiro(Tab) faça
4          se p_i ! = vazio então
5              continue;
6          fim
7          senão
8              node_struct * rx;
9              rx ← DataBaseModule.Get(p_i.move_number);
10             se rx ! = NULL então
11                 Double ADM_value ← GetAverage(rx);
12                 Double PK_value ← PriorKnowledge.Get(rx);
13                 p_i.move_value ← γ × PK_value + (1 − γ) × ADM_value;
14             fim
15             senão
16                 p_i.move_value ← PriorKnowledge.Get(rx);
17             fim
18             p_i.count_value ← (tabuleiro_pequeno?9 : 18);
19         fim
20     fim
21     TransfereValores( Tab, moves);
22 fim
```

Tal consulta se dá através do submódulo de Avaliação Final ADM, que por sua vez, consulta o submódulo de Banco de Dados e as heurísticas *Prior-Knowledge*, realizando um balanço entre o valor obtido pelos dois métodos.

Os algoritmos 5 e 6 ilustram como é realizado o processo de expansão no agente Go-Ahead.

No algoritmo 5 o fluxo é desviado para o módulo ADM de forma a mudar a avaliação de conhecimento prévio (linha 4).

O algoritmo 6 detalha o processo de avaliação de um movimento no agente Go-Ahead. Na linha 2 a variável *tabuleiro_pequeno* representa se o tabuleiro em jogo possui dimensões maiores ou menores que quinze. Analogamente ao processo no Fuego, o agente itera sobre todo o tabuleiro buscando as posições vazias a serem avaliadas (linha 3). Assim que

encontrada uma posição vazia é começado o processo de avaliação através de uma consulta no módulo de Banco de Dados ADM *DataBaseModule* sobre tal posição (linha 9). Se o módulo de Banco de Dados não possui nenhum registro sobre tal posição, é utilizado apenas o conhecimento prévio modelado pelo módulo de *PriorKnowledge* (linha 16).

Caso contrário, o algoritmo realiza um balanço, ponderado através da variável γ, entre os valores obtidos com o módulo ADM e os valores obtidos com as heurísticas de *PriorKnowledge* (linha 13). O método *GetAverage* mostrado na linha 11 retorna, simplesmente, a divisão entre o valor acumulado representado pelo atributo *move_value* e seu contador *move_count* de uma estrutura *node_struct*.

Após iterar sobre todas as posições vazias do tabuleiro, o processo é finalizado com a transferência de valores entre o tabuleiro e a lista de movimentos candidatos a serem expandidos (linha 21).

4.3.3.1 O parâmetro gama

Como mencionado anteriormente, o parâmetro γ é o responsável por gerar o balanço entre o valor estimado através das heurística $Prior - Knowledge$ e o valor estimado através do módulo ADM. Essa abordagem gera o levantamento de várias questões, entre elas:

1. Por que não utilizar apenas os valores gerados pelo módulo ADM?
2. Por que combinar os dois valores?
3. Por que utilizar uma variável de balanço?
4. Qual o melhor valor para o parâmetro γ?

Esses tópicos serão respondidos nos parágrafos seguintes.

Primeiramente, é importante ressaltar que o módulo ADM trabalha de forma completamente *online*. Sendo assim, todo conhecimento utilizado pelo módulo é gerado apenas em tempo de jogo. Em outras palavras, o agente não realiza nenhuma extração de informação previamente ao início de uma partida. E todo esse conhecimento é reiniciado ao começo de cada partida.

Dessa forma, a opção de utilizar apenas os dados gerados pelo módulo ADM em substituição à heurística *Prior-Knowledge* introduziria uma leve desvantagem na fase de abertura do jogo (até os 50 primeiros movimentos). No jogo de Go, a fase de abertura é crucial para o desenvolvimento das dinâmicas subsequentes. É nela em que se determina a estratégia e *frameworks* a serem utilizados. E como explicado previamente, o módulo ADM necessita de um volume de dados considerável para realizar as estimativas de forma mais assertiva. Por outro lado, ao início de cada partida o módulo ADM ainda não possui

dados suficientes para realizar tais estimativas em relação aos melhores movimentos, realizando, algumas vezes, jogadas sub-ótimas comprometendo assim a qualidade de abertura de jogo do agente e, consequentemente, seu desempenho na partida.

De toda forma, à medida que o jogo evolui, as estimativas realizadas pelo módulo ADM se tornam mais precisas e especificas ao jogo corrente, o que atribui a tal módulo uma alta capacidade de adaptabilidade e dinamismo. Sendo assim, torna-se natural a utilização de um balanço entre as duas técnicas em busca de uma combinação de suas maiores qualidades: a confiabilidade das estimativas realizadas pela heurística *prior-knownledge* em conjunção com a adaptabilidade e dinamismo oferecidos pelo módulo ADM.

A utilização de uma variável de balanço visa combinar dinamicamente a força entre o conhecimento gerado pelos dois módulos, criando assim, um equilíbrio entre especificidade de jogo e conhecimento supervisionado. Dessa forma ao se avaliar um movimento a ser selecionado durante a fase de expansão, leva-se em conta a posição e caráter geral de tal movimento, o que é avaliado através das regras estáticas estipuladas pelo módulo *Prior-Knowledge*, e a qualidade de um movimento para o cenário atual do tabuleiro, a qual é obtida por meios do módulo ADM.

A codificação do parâmetro γ como uma variável e não como uma constante teve o intuito de deixar o módulo preparado para uma futura investigação, a qual é modificar o algoritmo de modo que tal parâmetro se altere dinamicamente durante o jogo. Uma vez que o módulo ADM necessita de uma quantidade de dados maior para realizar boas estimativas, vale a pena investigar o desenvolvimento crescente de tal parâmetro, onde o mesmo começaria baixo no começo do jogo e aumentasse à medida que o jogo evoluísse.

Durante o desenvolvimento e testes do agente foram utilizados diversos valores para a variável γ, sendo assim os valores apresentados na seção 5 foram obtidos de forma exclusivamente empírica e uma análise detalhada será fornecida posteriormente.

4.3.4 O Módulo ADM na prática

A fim de para ilustrar um pouco mais o funcionamento do módulo ADM no agente Go-Ahead, considerar-se-á a mesma situação apresentada na Figura 11. Na figura em questão, o jogador das peças brancas possui a vez e necessita responder a formação chinesa elaborada pelas peças pretas.

Como mencionado anteriormente, uma estratégia eficaz, proposta por profissionais do jogo, é neutralizar a influência do *framework* das peças pretas sobre todo o tabuleiro. A figura 16 mostra uma análise de influência da jogada A.

Como pode-se perceber através da Figura 16, a jogada A além neutralizar, de certa forma, o *framework* estabelecido pelas peças pretas, também estabelece uma excelente conexão com as peças brancas à esquerda, criando também um *framework* e equilibrando a partida. Logo, para esse cenário, tal jogada se torna o objetivo do agente Go-Ahead.

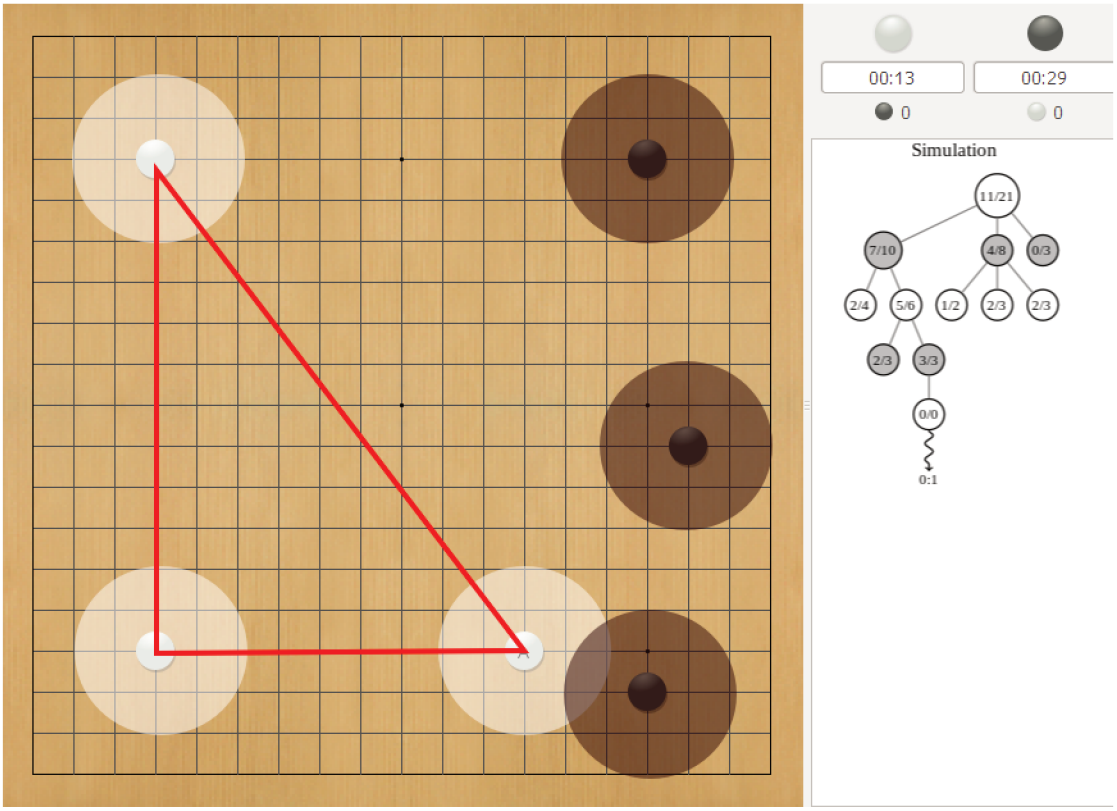

Figura 16 – Análise de influência da jogada A.

Para provar a eficiência do balanço entre o valor de conhecimento prévio e as estimativas online do módulo ADM, foi realizada uma investigação sobre a situação apresentada na figura 11. Como pode-se notar em tal figura, o valor atribuído pelo módulo *Prior-Knowledge* é de $0.4f$ para cada uma das quatro jogadas candidatas. Desse modo a escolha final de qual nó deverá ser inserido na árvore e posteriormente estimado através de uma simulação *Play-Out* se torna aleatória, uma vez que as estimativas são iguais.

O objetivo do agente é reduzir esse caráter aleatório utilizando informação gerada de forma *online*. Sendo assim, a tabela 3 mostra os valores finais após uma estimativa realizada pelo módulo ADM.

Tabela 3 – Resultado final das estimativas realizadas pelo módulo ADM.

Jogada	Valor *Prior-Knowledge*	Valor ADM	Valor Combinado
A	0.4	0.52	$0,424$
B	0.4	0.47	$0,414$
C	0.4	0.39	$0,398$
D	0.4	0.46	$0,412$

Como mostrado na tabela 3, os valores finais se diferenciam um do outro devido à combinação realizada com as estimativas fornecidas pelo módulo ADM.

Dessa forma, o agente Go-Ahead consegue deixar as estimativas um pouco mais particulares, uma vez que os valores agora são sutilmente diferenciados. Isso permite atenuar um pouco do caráter randômico do agente automático.

Vale ressaltar que no exemplo da Figura 11 apenas as jogadas A, B, C e D foram consideradas. Para isso, no grau de profundidade seis da árvore de busca, apenas tais nós foram levados em consideração. No experimento, o agente foi configurado para realizar cinquenta mil simulações por jogada. Apesar de o resultado para esse caso específico ser bastante positivo, ainda não permite assumir que o método de estimativas do módulo ADM seja capaz de substituir por completo as heurísticas *Prior-Knowledge*.

Uma análise detalhada da performance do agente e da relação do módulo ADM com as simulações *play-out* é fornecida na seções subsequentes.

4.4 Análise do impacto das simulações *play-out* no módulo ADM

Para elucidar o funcionamento do módulo ADM em conjunto com as simulações *Play-Out*, esta seção apresenta de forma detalhada uma análise do impacto da qualidade de tais simulações na qualidade das estimativas fornecidas pelo módulo ADM.

4.4.1 Política Aleatória nas simulações *Play-Out*

O algoritmo de busca MCTS, em seu modo original, tem o objetivo de qualificar tomadas de decisões através de amostragens aleatórias do domínio em questão. Assim como apresentado previamente, a ideia principal do algoritmo é montar de forma iterativa, uma árvore de busca de forma a otimizar as tomadas de decisões através do espaço de busca.

Contudo, tal política aleatória não é a opção ótima quando se trata de domínios em que o espaço de busca é gigantesco, como é o caso do jogo de Go. Muller mostrou que no agente Fuego, a inserção de apenas uma regra heurística na política de *play-out* se mostrou inquestionavelmente superior à política aleatória (FERNANDO; MÜLLER, 2013).

Esse fato pode ser facilmente concluído se pensarmos na qualidade das jogadas realizadas durante a fase de *Play-Out*. Tomemos uma situação onde exista apenas quatro jogadas a serem realizadas sendo elas A, B, C e D. Vamos supor que a partir de cada uma das quatro jogadas existam exatamente dez combinações possíveis de sequências em simulações de *Play-Out*, assim como ilustrado na Figura 17.

Através de uma política aleatória e com tempo de busca suficiente, seria possível executar as quarenta simulações possíveis a partir dos quatro nós. Agora, suponha que exista apenas um caminho considerado ótimo, ou seja, o caminho perfeito para tal situação, onde cada jogada é resultado da maximização de vantagem para cada jogador. Vamos consi-

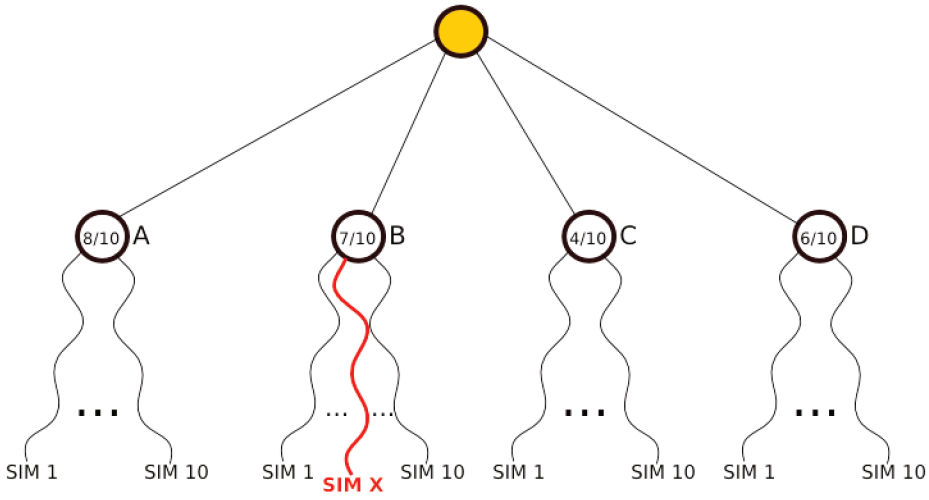

Figura 17 – Sequências *Play-Out*.

derar tal caminho representado pela $SIMX$ pertencente ao nó B na Figura 17. Assim como mostrado em tal figura, considere a porcentagem de ganho para cada jogada sendo representado pela razão localizada no interior de cada nó, ou seja, 80%, 70%, 40% e 60% para os nós A, B, C e D, respectivamente.

Como pode-se perceber, o algoritmo MCTS, utilizando uma política aleatória, não será capaz de tomar a decisão de escolher a jogada B, mesmo sendo a melhor jogada para o agente nessa situação específica. Isso se deve ao fato de que o algoritmo se baseia em quantidade e não na qualidade das simulações. Logo utilizando uma política aleatória e sem uma função de avaliação de qualidade das simulações *Play-Out* é impossível determinar qual a sequência mais forte e mais vencedora para o jogador em tal situação. Lembrando que, nessa situação, a sequência mais forte é aquela composta sempre por jogadas ótimas para os dois lados, a partir do estado corrente.

4.4.2 Qualidade das estimativas

A partir da análise conduzida até aqui, pode-se concluir que os fatores cruciais à alavancagem do algoritmo MCTS no jogo de Go incluem o algoritmo em si e a utilização de várias outras técnicas como RAVE, FPU, *prior-knowledge*, além de inúmeras heurísticas aplicadas às simulações *Play-Out*.

As estimativas realizadas pelo módulo ADM são bastante similares ao uso da técnica AMAF, inseridas no algoritmo RAVE. A principal diferença das estimativas ADM é que tal módulo utiliza toda a sequência *Play-Out* para gerar o conhecimento aplicando-o, posteriormente, às estimativas de conhecimento prévio em outras buscas. Esse fato nos permite concluir que a qualidade das estimativas ADM está diretamente relacionada à qualidade das simulações *Play-Out*.

Por exemplo, simulações pobres, ou seja, aquelas que são compostas por movimentos fracos durante toda a sequência, podem, de certa forma, poluir a qualidade das estima-

tivas, uma vez que movimentos considerados fracos podem ser associados inúmeras vezes com vitória aumentando assim sua qualidade na memória do agente.

Por outro lado, a política de simulação *Play-Out* apresentada pelo agente Fuego, se mostrou satisfatória e bem melhor do que a utilização de uma política apenas aleatória (FERNANDO; MÜLLER, 2013). Sendo assim, a qualidade das estimativas realizadas pelo módulo ADM será impactada, não somente pelas heurísticas de *Prior-Knowledge*, mas também pela qualidade das simulações realizadas na fase de *Play-Out*.

Para ilustrar tal cenário, será utilizado um caso prático apresentada na subseção seguinte.

4.4.3 Caso prático

Para ilustrar o impacto das simulações *play-out* na qualidade das estimativas realizadas pelo módulo ADM, considerar-se-á o cenário apresentado a seguir.

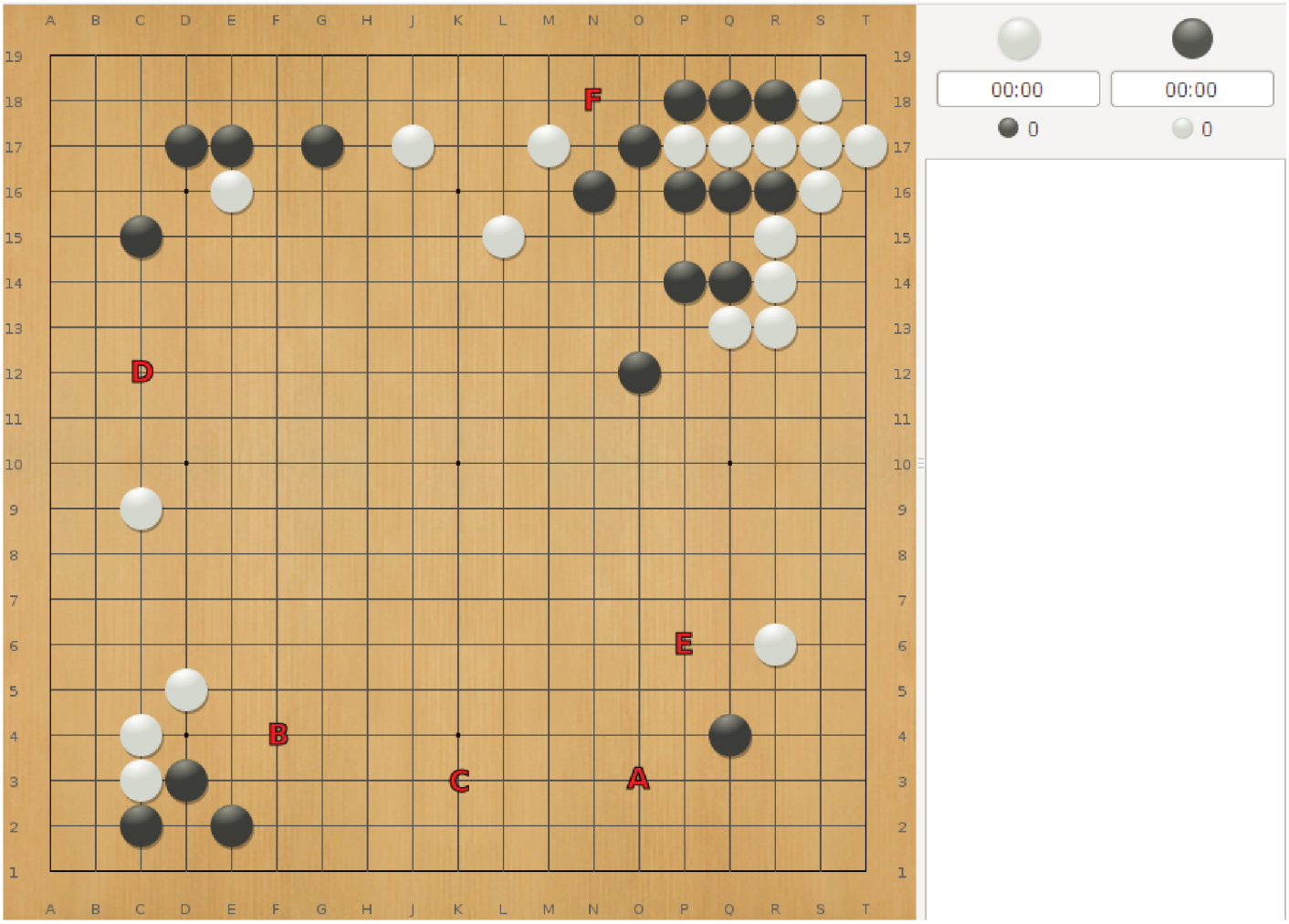

Figura 18 – Cenário jogada GRANDE X jogada URGENTE.

A fim de entender esse cenário um pouco mais complexo, é necessário introduzir o conceito de jogada grande e jogada urgente.

Uma jogada grande é aquela jogada que atribuí vários pontos, quase que de imediato, a um jogador específico. Em Go, é aquela jogada que permite ao jogador conquistar um espaço de território significativo. Se uma jogada em uma posição P permite a um jogador conquistar bastante território, ela é considerada grande. Se a jogada em uma mesma posição permite a ambos os jogadores conquistarem um território expressivo, ela é considerada muito grande. Por exemplo, na Figura 18, a jogada D permite a ambos os jogadores conquistarem um bom espaço de território. Apesar disso, ela é considerada apenas grande pois essa soma de território não é tão expressiva.

Uma jogada urgente é aquela jogada que não atribuirá ao jogador imediatamente uma quantidade expressiva de pontos, mas irá prevenir que tal jogador seja colocado em desvantagem, algumas vezes até o colocando em uma situação confortável para posteriores jogadas, servindo assim como um investimento a longo prazo.

No tabuleiro apresentado na Figura 18, as jogadas A, B, C, D e E são consideradas jogadas grandes, pois atribuem boas quantidades de território ou uma situação bem favorável ao jogador da vez.

No entanto, a jogada F é considerada uma jogada extremamente urgente, independente do jogador que possui a vez. Caso o a vez seja do jogador das peças brancas e ele realize tal movimento, o grupo composto pelas peças pretas na parte superior direita será colocado em risco, uma vez que será capaz de construir apenas um olho em *gote* (jogada em que o jogador perde a iniciativa), e passará, a partir desse momento a ser perseguido pelas peças brancas que será capaz de desenvolver ataques com propósitos múltiplos ameaçando o grupo preto e construindo território branco ao mesmo tempo. Sendo assim é quase natural dizer que as brancas teriam ganho o jogo. Esse cenário é ilustrado pela Figura 19.

Opostamente, caso a vez seja do jogador das peças pretas e ele realize tal jogada, ao mesmo tempo ele consegue estabilizar seu grupo evitando um possível severo ataque e agora inverte o jogo atacando e colocando em risco o grupo de peças brancas localizado na parte superior do tabuleiro. A partir daí é responsabilidade do jogador branco defender seu grupo concedendo assim a vantagem do ataque ao jogador preto. Tal cenário pode ser elucidado melhor través da Figura 20.

Sendo assim, foi simulado tal cenário considerando apenas as jogadas de A a F, para que um comparativo fosse realizado entre política aleatória e a política já adotada pelo agente Fuego.

O experimento foi executado cem vezes sendo que em cada experimento foram utilizadas cem mil simulações e logo após foi realizada uma estimativa ADM para comparar os valores adquiridos utilizando-se as duas políticas de simulação. Em tal experimento foi considerado que a vez fosse sempre do jogador preto. A Tabela 4 apresenta os resultados

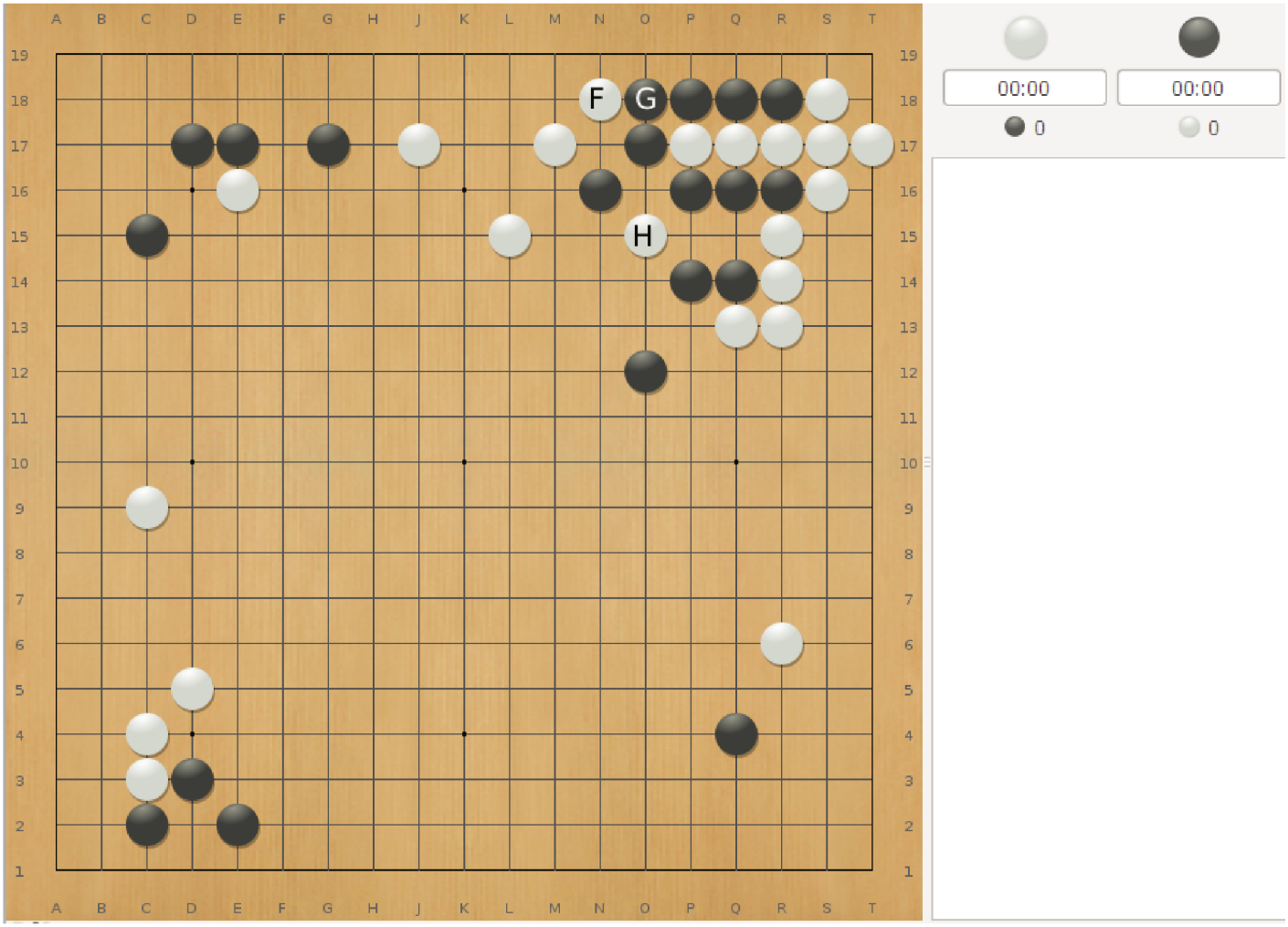

Figura 19 – Branco realiza a jogada F e desenvolve uma considerável vantagem sobre o jogador das peças pretas.

de tal experimento mostrando qual a porcentagem de escolha do movimento F através das duas políticas comparadas. No experimento foram considerados apenas os movimentos apresentados na Figura 18 para que um controle maior do ambiente pudesse ser adquirido.

Tabela 4 – Qualidade da escolha entre jogada grande x jogada urgente através de uma política aleatória e uma não aleatória.

Política	Porcentagem de escolha da jogada urgente F
Aleatória	24%
Não Aleatória	53%

Como pode ser notado na tabela 4, uma política não aleatória possibilitou ao módulo ADM estimar melhor, escolhendo assim a jogada F em 53% das execuções contra apenas 24% da política aleatória.

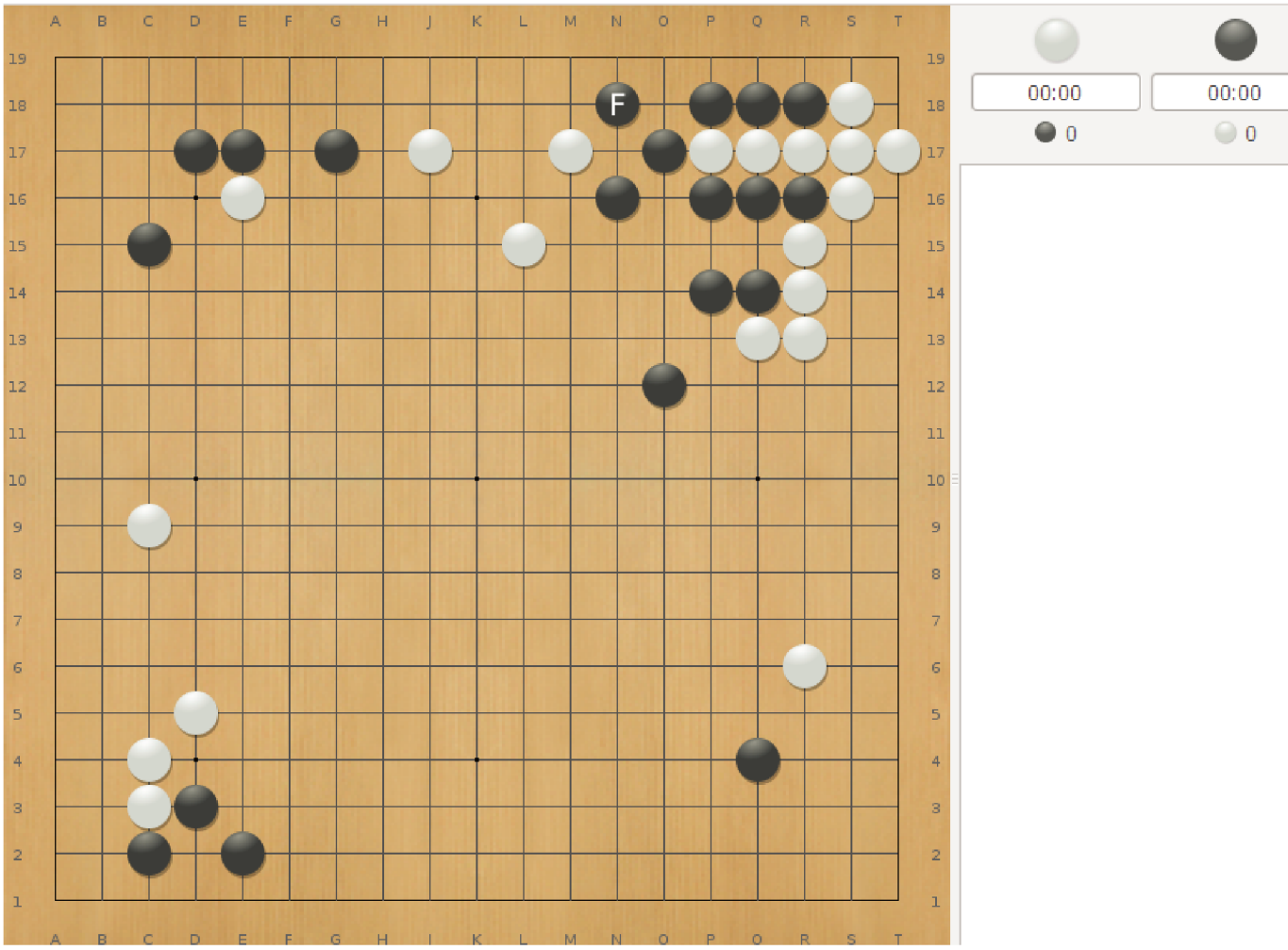

Figura 20 – Preto realiza a jogada F agora obtém a vantagem de atacar, colocando branco em uma situação desvantajosa.

Dessa forma, pode-se concluir que a qualidade da estimativa ADM está diretamente relacionada à qualidade das sequencias executadas através de uma simulação *Play-Out*. Assim, quanto melhor a política de simulação, melhor a estimativa do módulo ADM, já que o conhecimento desse último provém das simulações realizadas pelo agente.

É importante ressaltar que essa questão entre grande e urgente é um assunto de suma importância para a área de Inteligência Artificial, uma vez que tal dilema está presente em várias situações do cotidiano e envolve o conceito da maximização de escolhas ótimas em um agente inteligente. Por exemplo, imagine que um agente inteligente se encontre com fome e se depare com uma mesa onde há um banquete oferecido de forma gratuita, mas que ele necessite ir ao banheiro imediatamente. Comer o banquete oferecido gratuitamente é uma decisão grande, mas ir ao banheiro primeiro é uma decisão urgente.

A seção 5 apresenta os experimentos executados e resultados obtidos que comprovam o cumprimento dos objetivos da abordagem proposta neste trabalho.

Capítulo 5

Experimentos e Análise dos Resultados

Esta seção tem como objetivo apresentar os métodos utilizados para avaliar o agente Go-Ahead em suas variadas configurações, analisando empiricamente qual configuração foi mais bem sucedida, bem como as razões para tal. Discute-se aqui também o que pode ser feito como próximas melhorias de modo que versões futuras do agente possam superar esta que é abordada neste trabalho, bem como em outros agente bem sucedidos na área.

Todos os testes realizados a fim de averiguar o êxito obtido pela introdução das estimativas ADM no algoritmo MCTS foram realizados em um *hardware* com processador Intel Core 2 Quad 2.4 GHz e um total de 8 GB de memória RAM. Tais testes foram divididos em três fases com o objetivo de analisar perspectivas diferentes do impacto das melhorias realizadas:

1. Fase empírica para estimativa do melhor valor para o parâmetro γ.
2. Competições com o número de simulações pré-determinado.
3. Competições com o tempo de jogada pré-determinado.

Este capítulo está organizado da seguinte maneira:

- A seção 5.1 apresenta os métodos utilizados para validar os benefícios obtidos com a introdução das estimativas ADM no algoritmo MCTS implementado pelo agente Fuego. Também são descritas as medidas de avaliação, conjunto de parâmetros e os trabalhos com os quais a proposta foi comparada.
- A seção 5.2 apresenta os resultados de acordo com o que foi descrito na seção 5.1. Tal apresentação é realizada através de tabelas que discriminam todos os parâmetros utilizados nos experimentos e seus respectivos resultados.
- A seção 5.3 apresenta uma avaliação final dos resultados, onde são apontados acertos e limitações da proposta.

5.1 Método para a Avaliação

A fim de avaliar os benefícios e limitações impostos através da inserção do módulo ADM na estrutura arquitetural do agente Fuego, foram realizados vários testes em diversos cenários. Em todos os cenários apresentados na seção 5.2 a medida utilizada para avaliação do desempenho do agente desse trabalho é a taxa de vitória obtida sobre o agente Fuego através de torneios de 500 jogos por execução.

5.1.1 Sobre as configurações de tabuleiro

Primeiramente, para os testes realizados foram utilizadas duas configurações distintas de tabuleiros: 9x9 e 19x19. O propósito de utilizar tais configurações é avaliar o impacto das estimativas em diferentes espaços de busca. Várias técnicas já se mostraram eficientes para uma determinada configuração e não tão eficientes para outra. Por exemplo, alguns trabalhos relatam um grande benefício através da utilização de tabelas de transposição TT em configurações menores como a 9x9, mas, por outro lado, pouco impacto quando utilizada a configuração 19x19 (WERF; HERIK; UITERWIJK, 2003). Isso se deve ao fato de que o espaço de busca para configurações 9x9 é consideravelmente menor que o espaço de busca para configurações 19x19, aumentando assim a probabilidade de encontrar estados, transpostos ou não, anteriormente avaliados.

5.1.2 Sobre as configurações do jogo e plataformas de testes

Em cada torneio realizado durante os experimentos, o agente Go-Ahead joga, alternadamente, tanto com as peças brancas quanto com as pretas. Como mencionado previamente no capítulo 2, as peças pretas iniciam o jogo deixando as brancas em uma sutil desvantagem. Tal desvantagem é amenizada através da utilização do *komi*. Nos experimentos executados o valor de *komi* utilizado foi de $6,5$.

O jogador Go-Ahead foi testado apenas contra seu antecessor Fuego. Isso se deve ao fato de que dentre os jogadores que integram o estado da arte, o agente Fuego é o único que é distribuído de forma gratuita e com código aberto, o que facilita muito uma análise diligente de todo o processo envolvido. A versão do agente Fuego utilizada foi a versão corrente presente no repositório até a data de 3 de fevereiro de 2015 (ENZ; MULLER, 2015).

O programa utilizado para realizar os jogos entre os dois agentes foi o gogui-twogtp (GOGUI-TWOGTP, 2015) o qual funciona como uma interface para o protocolo *Go Text Protocol* (GTP) (GO..., 2015) permitindo a execução de jogos entre programas de Go. Dessa forma todos os testes foram automatizados tendo como árbitro o programa GNU-Go, ou seja, a avaliação e decisão final sobre o resultado da partida foi dada por um programa terceiro.

Ambas implementações, Fuego e Go-Ahead, foram configuradas para utilizar apenas uma *thread* durante o processo de busca. Isso facilitou a análise dos logs de ambos os programas a fim de realizar ajustes no agente Go-Ahead. Alguns jogos foram analisados minuciosamente de forma gráfica com o intuito de identificar jogadas consideradas muito ruins na tentativa de melhorar o agente. O programa utilizado para realizar tais análises gráficas foi o GoGui (GOGUI..., 2015).

5.2 Experimentos

Para avaliar o desempenho do agente Go-Ahead contra seu antecessor, três fases foram estabelecidas.

5.2.1 Fase para determinação empírica do valor γ

Como os testes levam bastante tempo a serem executados, fica praticamente inviável realizar os experimentos para todas as combinações de valores γ com as diferentes possibilidades de número de simulações.

Logo, essa fase consiste em estabelecer um número de simulações fixo, e variar o valor γ de modo a encontrar o melhor valor possível para a variável de balanço entre a estimativa realizada pela heurística *prior-knowledge* e aquela realizada pelo módulo ADM (regra de atualização 8). É importante ressaltar que o melhor valor encontrado nessa fase será utilizado nas fases seguintes onde há variações de número de simulações e tempo de jogada.

O número de simulações estabelecidas para tal experimento foi de 128.000 simulações por jogada. A tentativa aqui foi de estabelecer o maior número de simulações possíveis para que o módulo ADM tenha dados para trabalhar, uma vez que o mesmo trabalha com dados gerados pelas simulações.

Ao executar 128.000 simulações por jogada os experimentos se tornaram bastante lentos e a construção da árvore passou a ocupar um quantidade considerável de memória RAM, adicionando assim, ainda mais lentidão ao processo.

Para esta fase foram considerados cinco valores principais para a variável γ:

- 0: Ao assumir o valor 0 o agente executa sem a heurística *prior-knowledge*, utilizando puramente as estimativas do módulo ADM.
- 0.2: Atribuindo 0.2 ao valor γ, a ideia é permitir que o módulo ADM não opere sozinho mas tenha uma participação maior que as heurística *prior-knowledge*.
- 0.5: Com o valor de 0.5 o impacto de cada técnica é dividido igualmente.

- 0.8: Assumindo 0.8 como valor, a variável γ deposita grande parte da estimativa final na responsabilidade das heurísticas *prior-knowledge*, deixando para o módulo ADM, a responsabilidade de atenuar o caráter aleatório em estimativas igualitárias.
- 0.9: Atenua-se ainda mais o impacto das estimativas ADM na avaliação final do movimento.

As tabelas a seguir, mostram os resultados para os diferentes valores de γ. Em tais tabelas a primeira coluna representa o tipo de tabuleiro em que os testes foram executados, a segunda representa a taxa de vitória do agente Go-Ahead sobre o agente Fuego e a terceira representa o tempo extra total gasto em relação ao agente Fuego.

Vale ressaltar que cada linha das tabelas a seguir representa um torneio composto por 500 jogos onde cada agente assumia uma cor alternadamente.

Tabela 5 – Resultados para γ igual a 0.

Tabuleiro	Taxa de vitória do agente Go-Ahead	Tempo extra gasto
9x9	23%	2%
19x19	11%	5%

Como pode ser percebido através da tabela 5, a estratégia de substituição total da heurística *prior-knowledge* não se mostrou tão eficiente. Isso provavelmente se deve ao fato de que em alguns casos durante disputas locais por território, o módulo ADM não é capaz de encontrar o *tesuji* (melhor jogada local). Ao mesmo tempo isso prova que as verificações heurísticas de situações com *self-atari*, *atari* e análise de situações padrões do jogo, constituem um grande método de avaliação em geral. É notável também que a utilização do módulo introduziu uma limitação no tempo médio de busca aumentando-o em 2% para configurações 9x9 e 5% para configurações 19x19, em relação ao tempo gasto por seu adversário.

Tabela 6 – Resultados para γ igual a 0.3

Tabuleiro	Taxa de vitória do agente Go-Ahead	Tempo extra gasto
9x9	27%	4%
19x19	15%	7%

A tabela 6 mostra que ocorreu uma singela melhora de desempenho com a introdução do módulo de heurísticas na avaliação final do movimento. Ainda assim, tal melhoria é, ainda, muito modesta. Isso, provavelmente se deve ao fato do módulo ADM estar operando com maior peso comparado ao módulo heurístico. Há também um sutil aumento no tempo médio total gasto pelo agente Go-Ahead devido à introdução do módulo de heurísticas novamente.

Tabela 7 – Resultados para γ igual a 0.5

Tabuleiro	Taxa de vitória do agente Go-Ahead	Tempo extra gasto
9x9	46%	4%
19x19	39%	8%

Já a tabela 7 mostra um substancial ganho quando comparado com a Tabela 5. Como pode ser notado, o agente Go-Ahead já se torna competitivo em tabuleiros 9x9 e consegue uma margem melhor em 19x19. Isso, provavelmente, se deve ao fato de que falhas antes cometidas pelo módulo ADM são agora amenizadas pelo módulo heurístico. Pode-se notar também que o tempo extra utilizado pelo agente continua estável por volta de 4% a mais que seu adversário em tabuleiros 9x9 e 8% em tabuleiros 19x19.

Tabela 8 – Resultados para γ igual a 0.8

Tabuleiro	Taxa de vitória do agente Go-Ahead	Tempo extra gasto
9x9	58%	6%
19x19	53%	8%

A Tabela 8 reporta os melhores resultados encontrados nessa fase. Nas duas configurações o agente Go-Ahead foi capaz de superar seu adversário Fuego, conseguindo 58% de vitória em tabuleiros 9x9 e 53% em tabuleiros 19x19.

Tabela 9 – Resultados para γ igual a 0.95

Tabuleiro	Taxa de vitória do agente Go-Ahead	Tempo extra gasto
9x9	50%	5%
19x19	47%	7%

A tabela 9 reporta os resultados obtidos para com γ igual a 0.95. Apesar de bastante similares, os resultados não se mostraram melhores que aqueles obtidos com γ igual a 0.8 como mostrado na tabela 8. Isso provavelmente se deve ao fato de que ao minorar drasticamente a participação do módulo ADM, o agente passa a operar com as mesmas características do agente Fuego, todavia, com a incursão do custo imposto ao agente Go-Ahead. Em outras palavras, o agente Go-Ahead se torna o agente Fuego sutilmente limitado em termos computacionais.

Isso nos permite concluir, empiricamente, que o melhor desempenho do agente Go-Ahead sobre Fuego foi operando com um nível de autonomia 20% maior que seu oponente. Dessa forma 0.8 foi o valor adotado para a variável γ em todos os outros subsequente testes reportados a seguir.

5.2.2 Número de simulações pré-determinado

Essa fase se caracteriza por conter o número de simulações que cada agente realizará por jogada de forma pré-estabelecida. Isso é feito através do arquivo de configurações dos respectivos jogadores automáticos.

O objetivo de tal cenário é avaliar o desempenho do agente Go-Ahead em um ambiente livre de avaliação temporal. Ou seja, sem levar em consideração o tempo gasto para realizar o número de simulações pré-estabelecidas. Tal estratégia será capaz de mensurar e determinar o quão eficiente a abordagem proposta nessa trabalho foi em termos de majoração da autonomia do agente.

Essa fase dos experimentos foi dividida em quatro cenários diferentes, caracterizados pelo número de episódios presente em cada um deles:

- cenário I: 8000 episódios por jogada.
- cenário II: 16000 episódios por jogada.
- cenário III: 32000 episódios por jogada.
- cenário IV: 64000 episódios por jogada.

Tais cenários foram determinados, também, de forma empírica. O hardware e o tempo de execução dos testes também foram fatores cruciais na limitação da escolha do número de cenários. Com tais cenários, visa-se averiguar qualquer mudança de comportamento do agente à medida que o número de episódios por jogada aumenta.

Tabela 10 – Resultados para 8000 episódios por jogada.

Tabuleiro	Taxa de vitória do agente Go-Ahead	Tempo extra gasto
9x9	62%	7%
19x19	58%	8%

A tabela 10 apresenta os resultados obtidos para 8000 episódios de busca durante a escolha de uma jogada. Esses foram os melhores resultados obtidos durante o trabalho presente para ambas as configurações de tabuleiro. Esse fato é analisado ao final dessa subseção.

Tabela 11 – Resultados para 16000 episódios por jogada.

Tabuleiro	Taxa de vitória do agente Go-Ahead	Tempo extra gasto
9x9	56%	8%
19x19	54%	8%

A tabela 11 reporta os resultados obtidos quando pré-determinados 16000 episódios por decisão de movimento. Apesar de um pequeno decréscimo na taxa de vitórias, os resultados ainda são positivos para o agente Go-Ahead.

Tabela 12 – Resultados para 32000 episódios por jogada.

Tabuleiro	Taxa de vitória do agente Go-Ahead	Tempo extra gasto
9x9	56%	7%
19x19	52%	9%

Os resultados para a configuração de 32000 episódios por jogada, compondo o cenário III dessa fase, são apresentados na tabela 12.

Tabela 13 – Resultados para 64000 episódios por jogada.

Tabuleiro	Taxa de vitória do agente Go-Ahead	Tempo extra gasto
9x9	59%	8%
19x19	52%	10%

Finalmente, o número de episódios por jogada realizada é dobrado e, dessa forma, é concluído o cenário IV marcando também o encerramento dessa ase de experimentos.

5.2.2.1 Análise dos resultados obtidos

Essa fase dos experimentos marcou uma série de testes realizados com o propósito de averiguar o benefício obtido pelo agente desconsiderando o tempo de busca ao realizar uma jogada. A ideia principal aqui foi aumentar gradativamente o número de episódios realizados por jogada a fim de estabelecer algumas relações entre qualidade das estimativas com qualidade e quantidade das simulações *Play-Out*.

Como pode-se perceber, o melhor resultado obtido, para ambas as configurações de tabuleiro, pelo agente Go-Ahead no escopo desse trabalho foi apresentado de acordo com a tabela 10.

Esse resultado, provavelmente, se deve ao fato de que com um número menor de simulações a árvore de busca também se torna menor, necessitando ainda mais que seus nós sejam melhor selecionados. Apesar do número de simulações impactar diretamente na quantidade de resultados armazenados pelo módulo ADM, tal número não é o fator determinante da qualidade das estimativas realizadas pelo módulo. Inteligência e coerência na forma de conduzir as simulações realizadas na fase de *Play-Out* são fatores cruciais e determinantes na qualidade das estimativas fornecidas pelo módulo ADM.

Consequentemente, ao diminuir o número de simulações mantendo uma determinada qualidade na execução do *Play-Out*, grande parte da responsabilidade de boas jogadas é transferida à fase de pré-estimativas dos nós. Esse raciocínio completa-se ao inferir que a

minoração de escolhas aleatórias otimiza o processo de busca, uma vez que tal minoração é realizada baseada nos resultados obtidos em simulações. Essa relação é elucidada através do grafo apresentando pelo Figura 21.

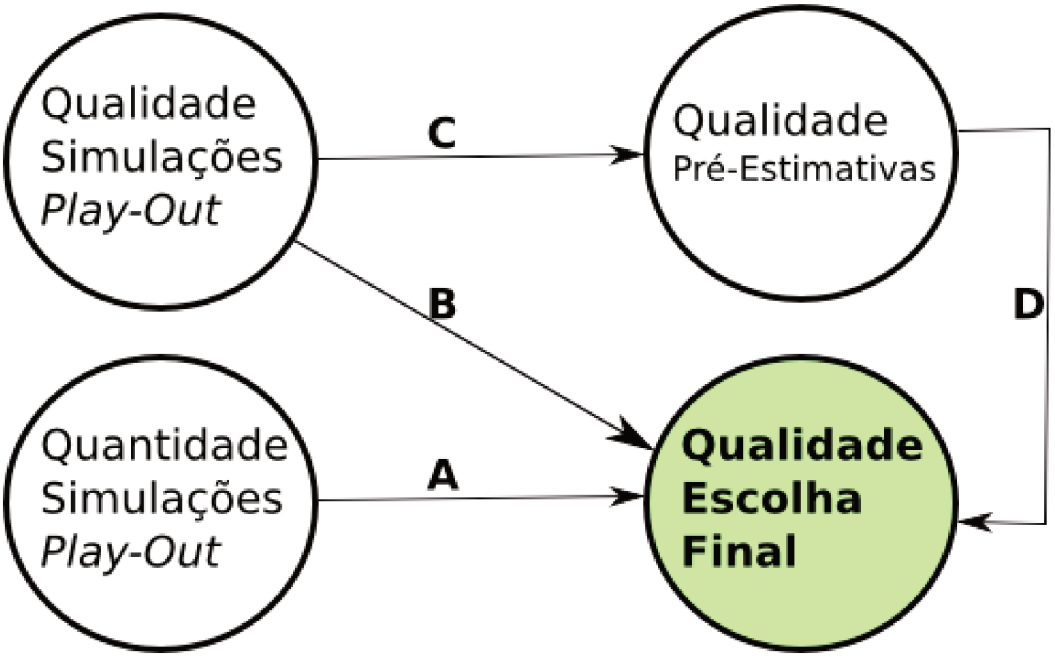

Figura 21 – Relação de influência entre as características analisadas no processo de busca MCTS.

Assim, ao se enfraquecer o arco **A**, cria-se uma dependência maior dos arcos **B** e **D**. Isso justifica a maior expressividade de resultados obtida pelo agente Go-Ahead quando utilizado um número relativamente baixo de episódios por jogada. O resultado disso tudo é um agente sutilmente melhorado pela capacidade de realizar boas pré-estimativas.

Analogamente, essa diferença é atenuada à medida que o número de episódios por jogada é acrescido. Como pode ser notado na tabela 13, o desempenho dos jogadores é extremamente equilibrado. Considerando a relação inferida, apresentada pela Figura 21, podemos concluir, naturalmente, que à medida que a quantidade de episódios é majorada, a dependência dos arcos **B** e **D** é também atenuada, levando assim a agentes de força mais equilibrada.

Dessa forma, a utilização do módulo ADM se mostrou bastante benéfica para essa fase dos experimentos. A seção a seguir introduz uma abordagem diferente de experimentos possibilitando uma análise através de novas perspectivas.

5.2.3 Competições com o tempo de jogada pré-determinado

O objetivo nessa fase dos experimentos foi observar e analisar o comportamento e performance do agente Go-Ahead tendo o seu tempo de realização de jogada limitado igualitariamente ao de seu oponente.

Dois limites de tempo foram analisados nessa fase:

- ❏ 10 segundos por jogada: esse é o padrão utilizado no agente Fuego, e permite que sejam executados ,em média, 70000 episódios de busca.

- ❏ 30 segundos por jogada: Foi configurado um tempo maior para que a performance do agente pudesse ser avaliada também com um tempo maior de processamento.

A tabela 14 apresenta os resultados obtidos para os testes executados com 10 segundos por jogada.

Tabela 14 – Resultados para os testes executados com tempo configurado para 10 segundos por execução de movimento.

Tabuleiro	Taxa de vitória do agente Go-Ahead	Défice em simulações (Go-Ahead)
9x9	52%	5%
19x19	50%	8%

Como pode ser observado através dos dados apresentados na tabela 14, o agente Go-Ahead ainda permanece competitivo com tempo limitado a 10 segundos por jogada. Enquanto o agente Fuego realiza cerca de 70000 episódios de busca por jogada na configuração 9x9 e 70000 na 19x19, seu oponente Go-Ahead realiza 5% e 8% menos episódios em tais configurações, respectivamente. Isso, naturalmente, é atribuído ao custo computacional inserido no novo agente.

Tabela 15 – Resultados para os testes executados com tempo configurado para 30 segundos por execução de movimento.

Tabuleiro	Taxa de vitória do agente Go-Ahead	Défice em simulações (Go-Ahead)
9x9	48%	6%
19x19	44%	10%

A tabela 15 apresenta os resultados para testes com tempo limitado a 30 segundos por jogada. Nessa abordagem dos testes o agente Go-Ahead encontrou algumas limitações, não conseguindo assim superar seu oponente Fuego. Ao aumentar a quantidade de tempo por jogada aumenta-se também o número de simulações que o agente Fuego realiza aumentando assim seu poder. Isso, provavelmente, se deve ao fato de que na fase de abertura quando aumentado o número de episódios o agente Fuego realiza movimentos mais concisos, aumentando assim sua vantagem desde o começo do jogo.

5.3 Avaliação Final dos Resultados

Essa seção fornece uma visão final sobre os resultados obtidos na fase de experimentos. É importante ressaltar que todos os experimentos foram conduzidos sem o auxílio da técnica de livro abertura. Além do mais, é de extrema complexidade estabelecer a relação entre a performance do agente e cada técnica particular utilizado pelo mesmo, tais como, livro de abertura, Urgência de Primeira Jogada, RAVE, etc.

5.3.1 Benefícios oberservados

Um dos objetivos definidos neste trabalho de Mestrado foi a majoração da autonomia do agente Fuego. Isso foi alcançado com a atenuação da participação das heurísticas *prior-knowledge* no processo de pré-estimativa de um movimento a ser inserido na árvore de busca MCTS durante a fase de expansão.

Nesse sentido, foram realizados alguns testes em modo *debug* e também habilitada a escrita em *logs* para que pudesse ser averiguado em quantas das vezes o agente Go-Ahead utilizaria o módulo ADM para realizar uma distinção das estimativas dos movimentos gerados.

Ao realizar tal análise foi constatado que o módulo ADM foi realmente utilizado em 85% das vezes. Isso foi concluído observando-se a quantidade de pré-estimativas total e a porção das que foram mudadas através da combinação com os dados do módulo ADM.

É de extrema complexidade estimar em quantas das vezes essa distinção foi benéfica e quantas não, uma vez que para isso, deveria ser feita uma análise tático-estratégica profunda de cada jogada realizada pelo agente e o impacto do módulo ADM em cada decisão. Através dos resultados obtidos estima-se que tal modificação foi mais benéfica ao agente.

Outro benefício é a utilização dos dados de simulação *Play-Out*, antes descartados ao final de cada simulação, para geração de conhecimento, adicionando assim maior adaptabilidade ao agente Go-Ahead. Esse caráter dinâmico possibilitou ao agente sobressair seu adversário em situações com poucas simulações, o que pode ser favorável em situações de partidas *blitz*.

5.3.2 Limitações do módulo ADM

Apesar do benefício comprovado através dos resultados, foram constatadas algumas limitações no Go-Ahead, tais como:

- Avaliação singular: Os movimentos no módulo Go-Ahead, são avaliados independentemente do contexto geral em que se encontram. Uma boa abordagem para atacar tal limitação na fase de pré-estimativa é realizar também uma avaliação pela

resposta de algumas jogadas, assim como é feito em (BAIER; DRAKE, 2011), ou até mesmo gerar avaliações por *josekis* ou grupos de movimentos.

- Valor constante de γ: Como descrito previamente, o valor de γ permanece constante durante todo o confronto. Isso pode não ser uma boa abordagem, levando em consideração que o módulo ADM pode ser melhor ou pior em uma determinada fase do jogo. Uma boa solução para essa irregularidade seria programar a variável γ de forma auto-ajustável, atribuindo um peso maior ao módulo ADM na fase inicial do jogo e diminuindo seu impacto ao longo do tempo, por exemplo.

- Desconsideração temporal: Durante a realização de uma pré-estimativa dos nós a serem inseridos, todos as jogadas já aparecidas em algum *Play-Out* são consideradas, assim um movimento pode aparecer várias vezes como uma boa opção na fase de abertura do jogo e obter um valor tão alto capaz de ofuscar a qualidade dos movimentos realizados na fase de *yose*(fim de jogo). Uma boa alternativa seria gerar as estimativas baseadas nos últimos 100000 episódios, por exemplo, em uma tentativa de aumentar a capacidade de adaptação do agente.

- Sequência fixa da política de *Play-Out*: Durante a fase de simulação *Play-Out*, o agente obedece uma série de regras de movimentos estipulados através de uma política de *Play-Out*. Essa política foi manualmente inserida no código-fonte do agente, o que lhe torna uma solução pouco dinâmica. Dessa forma, pode acontecer de algumas simulações serem bastante similares uma vez que obedecem sempre a mesma gama regras. Por exemplo, imagine que na situação corrente do tabuleiro exista uma situação de *atari*, de acordo com a política de *Play-Out*, praticamente toda simulação iria iniciar por esse movimento, evitando assim que outros movimentos tivessem ao menos a chance de serem avaliados. O pesquisador também brasileiro Marcolino, propôs em seu trabalho (MARCOLINO; MATSUBARA, 2011) uma abordagem interessante para tratar tal problema. A ideia seria utilizar uma política auto-ajustável, ou até mesmo de ordem randômica durante a execução das simulações. Dessa forma teríamos simulações bem mais diversificadas o que aumentaria o poder de avaliação do módulo ADM.

O próximo capítulo apresenta a conclusão do trabalho e a descrição das próximas etapas a serem realizadas.

Capítulo 6

Conclusão

Neste trabalho foi apresentado o agente Go-Ahead, um jogador automático para o jogo de Go desenvolvido na linguagem C++ sobre o código-fonte do agente Fuego.

O jogo de Go foi o domínio escolhido por conter um enorme espaço de busca o que o torna um desafio gigante para as técnicas de inteligência artificial modernas. Além do mais, o jogo possui algumas sutilezas que o tornam ainda mais complexo como:

1. Decisão Local x Decisão Global: Muitas das vezes, o jogador deve avaliar o contexto global e decidir se deve ou não abrir mão de uma batalha local. Tal fato várias vezes favorece o acontecimento do chamado efeito horizonte, onde uma jogada pode se apresentar boa localmente mas muito ruim a longo prazo.

2. Decisão estratégica: Assim que o jogo se inicia o jogador tem que tomar algumas decisões estratégicas como se vai jogar territorial ou por influência, se vai adotar uma postura mais agressiva ou mais passiva, etc.

3. Escadas (*ladders*) e anti-escadas (*ladder-breakers*): O jogador tem que saber prever a formação de escadas para realizar jogadas anti-escadas, por exemplo.

4. Ameaças em KO: Em disputas de KO é muito importante que o jogador saiba identificar determinados padrões que o possibilitarão avaliar quais jogadas são importantes e quais não.

Todos esse conceitos estão relacionados a situações em que a tomada de decisões inteligentes é crucial ao desempenho do jogador na partida. Dessa forma, não basta apenas prever e realizar boas jogadas locais ou a curto prazo, é necessário ter uma visão que objetive maximizar a recompensa de uma forma global, assemelhando-se a várias situações cotidianas, como por exemplo, investimentos financeiros.

O algoritmo MCTS é uma das técnicas de inteligência artificial mais utilizada para atacar tal problema realizando buscas iterativas no espaço de busca do jogo. Através de

simulações até o final do jogo e com relativamente pouco conhecimento heurístico ele cria uma árvore busca objetivando sempre a melhor jogada corrente.

Uma das partes importantes na busca realizada pelo algoritmo MCTS é a fase de expansão, onde um nó é inserido na árvore de busca na esperança de ser avaliado através de uma simulação *Play-Out*. O agente Fuego implementa tal processo com algumas otimizações propostas por diversos pesquisadores ao redor do mundo e ao longo da história. Uma dessas otimizações é a avaliação de conhecimento prévio, onde busca-se inicializar um nó com valores confiáveis antes de sua inserção na árvore, o que por sua vez irá acelerar o processo de busca aumentando sua performance de forma geral.

O agente Go-Ahead modifica o agente Fuego através da utilização de estimativas semelhantes à técnica AMAF, buscando atenuar o caráter altamente supervisionado encontrado em seu antecessor através da utilização dos dados de simulações *Play-Out* antes descartados ao final de cada episódio.

A seguir são descritas as principais contribuições provenientes desse trabalho.

6.1 Principais Contribuições

A massiva quantidade de dados já simulados pode ser fonte de uma grande quantidade de conhecimento a respeito do jogo corrente. Isso combinado com uma análise diligente do processo de expansão contido no algoritmo MCTS motivou o autor desse trabalho a desenvolver o agente Go-Ahead, que faz uso de tais dados para melhorar o processo de expansão. Com isso, obtiveram-se significativos resultados na tarefa de melhorar o desempenho do agente e, paralelamente, conceder-lhe mais autonomia através da atenuação do uso de heurísticas.

Logo, as principais contribuições providas pelo jogador Go-Ahead são:

1. Melhoria de performance geral: Como apresentados na seção 5, os resultados comprovaram que, de uma forma geral, a utilização do módulo ADM melhorou a performance do jogador de go, principalmente em situações de partidas *blitz*.

2. Atenuação do caráter supervisionado: Através da extração do conhecimento embutido nas simulações *Play-Out*, o módulo ADM contribuiu para o aumento da autonomia do jogador automático com louvor, uma vez que o impacto da heurística *prior-knowledge* foi minorado.

Através das análises e experimentos realizados, foi possível responder às questões levantadas na seção 1.2, mostrando que:

- o modelo de pré-avaliação utilizado pelo agente Fuego não é ótimo e pode ser relativamente melhorado.

- ❏ O modelo antes totalmente estático foi sutilmente flexibilizado através da dinamicidade inserida com as estimativas do módulo ADM.

- ❏ A massiva quantidade de dados gerados durante as simulações *Play-Out* pode ser transformada em conhecimento útil e de qualidade contribuindo para futuras decisões.

A seção subsequente destaca os próximos passos a serem seguidos na continuação da pesquisa.

6.2 Trabalhos Futuros

Apesar dos resultados favoráveis apresentados, existem várias situações ainda a serem analisadas, o que gera assim, uma série de melhorias a serem investigadas tais como as descritas a seguir:

- ❏ Valor γ adaptativo: O valor γ que introduz o balanço entre o impacto causado pelo módulo ADM e aquele causado pelas heurísticas *prior-knowledge* foi mantido constante, como visto anteriormente. Serão realizadas algumas análises para identificar as mudanças de comportamento do agente de acordo com a variação *online* de tal parâmetro, podendo assim identificar com confiança em que fase do jogo o módulo ADM tem maior impacto.

- ❏ Divisão temporal das estimativas: Quando o movimento em uma posição P é simulado no início do jogo e, consequentemente ou ocasionalmente, isso leva a vitórias em simulações, tal movimento adquire uma boa estimativa. Acontece que muitas das vezes tal estimativa é valida para uma determinada fase do jogo, por exemplo, movimento que são bons somente na abertura do jogo. Contudo, como a memória ADM não engloba o conceito temporal do jogo, ela mantém a estimativa de uma forma geral, logo ao final do jogo mesmo quando tal jogada não for tão eficiente, ela poderá ter uma boa estimativa graças a simulações ocorridas no começo do jogo. Levando isso em consideração, o autor realizará uma investigação criando *clusters* de linha de tempo, onde cada *cluster* representará uma determinada fase do jogo.

- ❏ Modelo variável de política *Play-Out*: Assim como proposto por Marcolino em (MARCOLINO; MATSUBARA, 2011), um modelo variável, em que a ordem das regras avaliadas na política de *Play-Out* não seja fixa, pode trazer benefícios ao agente de uma forma geral. Isso acontece devido ao fato de que tal dinamismo aplicado à ordem das regras possibilitará uma maior variedade de simulações aumentando assim a qualidade das estimativas. Combinada com o módulo ADM tal abordagem pode ser ainda mais benéfica, uma vez que o aumento na variedade de movimentos

nas simulações ocasionará um aumento na variedade de movimentos inseridos na memória ADM, melhorando assim seu caráter avaliativo.

- Análise de Influência: Apesar dos resultados positivos obtidos no escopo deste trabalho, as estimativa podem ser expressivamente enriquecidas através de informações específicas do jogo e análises em tempo real. Por exemplo, a análise de influência de cada peça no tabuleiro provê muita informação importante, principalmente em âmbito estratégico, como mostrado na figura 22.

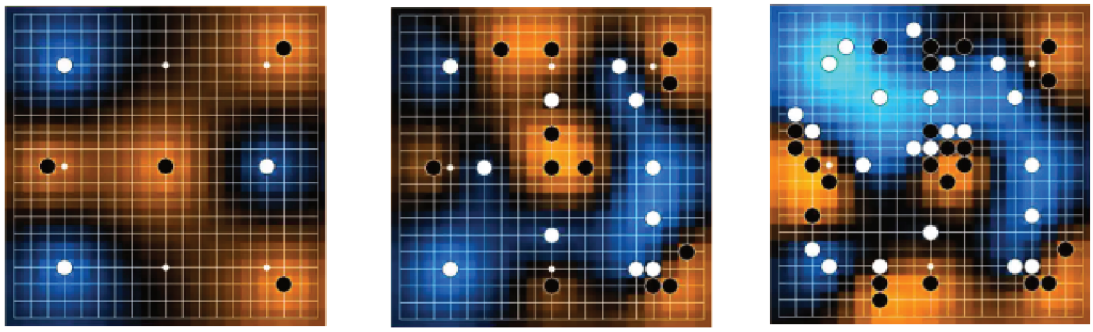

Figura 22 – Sequência de jogadas realizadas em uma partida de Go. Na imagem são exibidos os campos de influência de cada peça no tabuleiro, o que permite uma análise mais clara do jogo.

- Geração regras através de RBC: As regras utilizadas pela heurística *prior-knowledge* foram concebidas através de experimentos contra o jogador MoGo e observações com jogadores humanos. O autor desse trabalho já iniciou uma pesquisa onde realizará a extração de regras a serem analisadas através da técnica de geração de regas baseadas em casos (RBC) sobre uma base de dados de jogos profissionais nomeada GoGoD. Com isso, objetiva-se a geração de regras mais sofisticadas gerando um módulo de heurísticas *prior-knowledge* mais forte do que o atual utilizado no Go-Ahead e Fuego.

Referências

AN, D. **Handbook of Chinese mythology**. [S.l.]: Abc-clio, 2005.

ARNESON, B.; HAYWARD, R.; HENDERSON, P. Wolve wins hex tournament. **Icga Journal**, 2008.

BAIER, H.; DRAKE, P. D. The Power of Forgetting : Improving the Last-Good-Reply Policy in Monte Carlo Go. v. 2, n. 4, p. 303–309, 2011. Disponível em: <https://doi.org/10.1109/TCIAIG.2010.2100396>.

BRÜGMANN, B. **Monte-Carlo Go**. [S.l.], 1993.

BUMP, D. **Gnugo home page**. 2003. <https://www.gnu.org/software/gnugo/>. [Online; accessed 25-Feb-2015].

CHASLOT, G. **Monte-Carlo tree search**. Tese (Doutorado) — PhD thesis, Maastricht University, 2010.

CHASLOT, G. M. J.-b. **Monte-Carlo Tree Search door**. [S.l.: s.n.], 2010. ISBN 9789085590996.

CHILDS, B. E.; BRODEUR, J. H.; KOCSIS, L. Transpositions and move groups in monte carlo tree search. In: IEEE. **Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On**. 2008. p. 389–395. Disponível em: <https://doi.org/10.1109/CIG.2008.5035667>.

CORMEN, T. H. et al. **Introduction to algorithms**. [S.l.]: MIT press Cambridge, 2001. v. 2.

COULOM, R. Computing elo ratings of move patterns in the game of go. In: **Computer games workshop**. [s.n.], 2007. Disponível em: <https://doi.org/10.3233/ICG-2007-30403>.

_____. Efficient selectivity and backup operators in Monte-Carlo tree search. In: **Computers and games**. Springer, 2007. p. 72–83. Disponível em: <https://doi.org/10.1007/978-3-540-75538-8_7>.

DOMSHLAK, C.; FELDMAN, Z. To UCT, or not to UCT? (Position Paper).

ENGELBRECHT, A. P. **Computational intelligence: an introduction**. [S.l.]: John Wiley & Sons, 2007.

ENZ; MULLER. **Fuego Project**. 2015. <http://sourceforge.net/projects/fuego/>. [Online; accessed 25-Feb-2015].

ENZENBERGER, M. et al. Fuego—an open-source framework for board games and Go engine based on Monte-Carlo tree search. **Computational Intelligence and AI in Games, IEEE Transactions on**, IEEE, v. 2, n. 4, p. 259–270, 2010. Disponível em: <https://doi.org/10.1109/TCIAIG.2010.2083662>.

FERNANDO, S.; MÜLLER, M. Analyzing simulations in monte carlo tree search for the game of go. 2013. Disponível em: <https://doi.org/10.1007/978-3-319-09165-5_7>.

FINSLAB. **Dan adn Kyu**. 2015. <http://finslab.com/enciclopedia/letra-d/dan.php>. [Online; accessed 25-Feb-2015].

FR, I. **Comparação Ranking**. 2015. <http://www.inria.fr/en/>. [Online; accessed 25-Feb-2015].

GELLY, S. et al. The grand challenge of computer go: Monte carlo tree search and extensions. **Communications of the ACM**, ACM, v. 55, n. 3, p. 106–113, 2012. Disponível em: <https://doi.org/10.1145/2093548.2093574>.

GELLY, S.; SILVER, D. Combining online and offline knowledge in UCT. In: ACM. **Proceedings of the 24th international conference on Machine learning**. 2007. p. 273–280. Disponível em: <https://doi.org/10.1145/1273496.1273531>.

_____. Monte-Carlo tree search and rapid action value estimation in computer Go. **Artificial Intelligence**, Elsevier, v. 175, n. 11, p. 1856–1875, 2011. Disponível em: <https://doi.org/10.1016/j.artint.2011.03.007>.

GELLY, S.; WANG, Y. Exploration exploitation in go: Uct for monte-carlo go. In: **NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop**. [S.l.: s.n.], 2006.

GELLY, S. et al. Modification of UCT with patterns in Monte-Carlo Go. 2006.

_____. Modification of UCT with Patterns in Monte-Carlo Go. n. November, 2006.

GO Text Protocol Specification. 2015. <http://www.lysator.liu.se/~gunnar/gtp/gtp2-spec-draft2/gtp2-spec.html>. [Online; accessed 25-Feb-2015].

GOGUI project page. 2015. <http://gogui.sourceforge.net/>. [Online; accessed 25-Feb-2015].

GOGUI-TWOGTP. 2015. <http://gogui.sourceforge.net/doc/reference-twogtp.html>. [Online; accessed 25-Feb-2015].

GRID5000. **GRID5000**. 2015. <https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home>. [Online; accessed 25-Feb-2015].

HELMBOLD, D. P.; PARKER-WOOD, A. All-moves-as-first heuristics in monte-carlo go. In: **IC-AI**. [S.l.: s.n.], 2009. p. 605–610.

HUNTER, D. R. Mm algorithms for generalized bradley-terry models. **Annals of Statistics**, JSTOR, p. 384–406, 2004.

JENSEN, A. R. The relationship between learning and intelligence. **Learning and Individual Differences**, Elsevier, v. 1, n. 1, p. 37–62, 1989. Disponível em: <https://doi.org/10.1016/1041-6080(89)90009-5>.

JUNIOR, E. V.; JULIA, R. M. Btt-go: An agent for go that uses a transposition table to reduce the simulations and the supervision in the monte-carlo tree search. In: **The Twenty-Seventh International Flairs Conference**. [S.l.: s.n.], 2014.

KIERULF, A. **Smart game board: A workbench for game-playing programs, with Go and Othello as case studies**. Tese (Doutorado) — Swiss Federal Institute of Technology, 1990.

KISHIMOTO, A. **Correct and efficient search algorithms in the presence of repetitions**. Tese (Doutorado) — University of Alberta, 2005.

KOCSIS, L.; SZEPESVÁRI, C. Bandit based Monte-Carlo planning. In: **Machine Learning: ECML 2006**. Springer, 2006. p. 282–293. Disponível em: <https://doi.org/10.1007/11871842_29>.

KORF, R. E. Linear-space best-first search. **Artificial Intelligence**, Elsevier, v. 62, n. 1, p. 41–78, 1993. Disponível em: <https://doi.org/10.1016/0004-3702(93)90045-D>.

KSG$_SERVER$.**KGS home page**.2015.<>. [Online; accessed 25-Feb-2015].

LEE, C.-S. et al. The computational intelligence of mogo revealed in taiwan's computer go tournaments. **Computational Intelligence and AI in Games, IEEE Transactions on**, IEEE, v. 1, n. 1, p. 73–89, 2009. Disponível em: <https://doi.org/10.1109/TCIAIG.2009.2018703>.

LIN, G. I. Fuego Go: the missing manual. p. 1–18, 2009.

MARCOLINO, L. S.; MATSUBARA, H. Multi-Agent Monte Carlo Go. p. 21–28, 2011.

MÜLLER, M. **Computer Go as a sum of local games: an application of combinatorial game theory**. Tese (Doutorado) — Swiss Federal Institute OF Technology Zürich, 1995.

MÜLLER, M. Computer go. **Artificial Intelligence**, Elsevier, v. 134, n. 1, p. 145–179, 2002. Disponível em: <https://doi.org/10.1016/S0004-3702(01)00121-7>.

RUSSELL, S.; NORVIG, P. **Artificial Intelligence: A New Approach**. [S.l.]: Prentice Hall, NJ, 1995.

SCHAEFFER, J. The history heuristic and alpha-beta search enhancements in practice. **Pattern Analysis and Machine Intelligence, IEEE Transactions on**, IEEE, v. 11, n. 11, p. 1203–1212, 1989. Disponível em: <https://doi.org/10.1109/34.42858>.

SENSEI, L. **Compara Ranking**. 2015. <http://senseis.xmp.net/?Sh%2Frankcomparison>. [Online; accessed 25-Feb-2015].

_____. **Go Ranking**. 2015. <http://senseis.xmp.net/?Rank>. [Online; accessed 25-Feb-2015].

SHANNON, C. E. A chess-playing machine. **Scientific American**, v. 182, n. 2, p. 48–51, 1950. Disponível em: <https://doi.org/10.1038/scientificamerican0250-48>.

SILVER, D.; SUTTON, R. S.; MÜLLER, M. Reinforcement learning of local shape in the game of go. In: **IJCAI**. [S.l.: s.n.], 2007. v. 7, p. 1053–1058.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de Dados e seus Algoritmos**. [S.l.]: Livros Técnicos e Científicos, 1994. v. 2.

TESAURO, G. **Practical issues in temporal difference learning**. Springer, 1992. Disponível em: <https://doi.org/10.1007/978-1-4615-3618-5_3>.

TURING, A. et al. Can automatic calculating machines be said to think?(1952). **B. Jack Copeland**, p. 487, 2004. Disponível em: <https://doi.org/10.1093/oso/9780198250791.003.0020>.

TURING, A. M. Computing machinery and intelligence. **Mind**, JSTOR, p. 433–460, 1950. Disponível em: <https://doi.org/10.1093/mind/LIX.236.433>.

WASSERMAN, L. **All of statistics: a concise course in statistical inference**. Springer, 2004. Disponível em: <https://doi.org/10.1007/978-0-387-21736-9>.

WERF, E. C. van der; HERIK, H. J. V. D.; UITERWIJK, J. W. Solving go on small boards. **ICGA Journal**, Citeseer, v. 26, n. 2, p. 92–107, 2003. Disponível em: <https://doi.org/10.3233/ICG-2003-26205>.