ULYSSES BONFIM DOS SANTOS

GENERAL GAME PLAYING UTILIZANDO SIMULAÇÕES

Monografia apresentada ao curso de Ciência da Computação, Setor de Ciências Exatas, Universidade Federal do Paraná, como requisito parcial para a conclusão do curso.
Orientador: Prof. Dr. Marcos Alexandre Castilho

CURITIBA

2010

ULYSSES BONFIM DOS SANTOS

GENERAL GAME PLAYING UTILIZANDO SIMULAÇÕES

Monografia apresentada ao curso de Ciência da Computação, Setor de Ciências Exatas, Universidade Federal do Paraná, como requisito parcial para a conclusão do curso.
Orientador: Prof. Dr. Marcos Alexandre Castilho

CURITIBA

2010

SUMÁRIO

RESUMO

General Game Playing (GGP) é um campo da Inteligência Artificial que trabalha com sistemas que, dada a descrição de um jogo arbitrário, sabem joga-lo de maneira inteligente. Este é um problema bem mais complicado que construir um jogador para um jogo específico, pois não se pode contar com qualquer tipo de conhecimento sobre o domínio do jogo para gerar uma heurística.

Neste trabalho será apresentado uma abordagem para a construção deste sistema. A implementação é fundamentada em simulações para decidir qual o melhor caminho, utilizando o algoritmo de UCT baseado na técnica de Monte Carlo.

CAPÍTULO 1

INTRODUÇÃO

General Game Playing (GGP) é uma nova área de pesquisa na Inteligência Artificial, cujo o desafio está em construir um agente autônomo que consiga jogar efetivamente jogos que nunca viu antes. Ao contrario dos jogadores clássicos, especializados em apenas um jogo, como xadrez ou damas, as propriedades do jogo são desconhecidas pelo programador durante a construção do sistema. Isso demanda o uso e integração de várias técnicas da Inteligência Artificial, o que torna GGP ideal para o desenvolvimento de novos métodos. Desde de 2005, o *Stanford Logic Group* organiza uma competição para incentivar pesquisas nesta área. Patrocinada pela *American Association for Artificial Intelligence*, este evento oferece a oportunidade de comparar diferentes abordagens em um cenário competitivo.

Para demonstrar inteligência, os jogadores devem realizar uma série de movimentos que levam a uma posição final vitoriosa. As decisões de qual ação deve ser tomada são buscadas na árvore do jogo com auxílio de uma função de avaliação. Sistemas construidos para jogos específicos usam técnicas como livros de abertura e banco de dados com posições de fim de jogo para incrementar a função de avaliação. O campeão mundial de xadrez Deep Blue [2] tem um livro de abertura de mais de 4.000 posições e um sumário de 700.000 jogos de mestres do xadrez. Chinook [8], campeão mundial em damas, tem informações de mais de 443 bilhões de posições finais de jogo. A eficácia da função de avaliação impacta diretamente na busca por bons movimentos, permitindo que o sistema gaste mais tempo em áreas promissoras e menos em jogadas ruins.

Jogadores em GGP, no entanto, devem ser capazes de demonstrar inteligência mesmo em jogos que nunca tenha jogado. O problema central é construir uma função de avaliação, ou *heurística*, que seja genérica o suficiente para funcionar em qualquer jogo que ele venha confrontar. Mesmo que um sistema tenha acesso a heurísticas para jogos específicos, ainda seria preciso determinar qual, ou quais, seriam aplicáveis para o atual problema

enfrentado.

A competição, apesar de ser notariamente recente, mostra alguns caminhos para contornar o problema em gerar uma heurística. Fluxplayer[9] usa lógica Fuzzy para determinar o grau em que a posição atual satisfaz logicamente uma condição de vitória. Outra possibilidade é a extração de características comuns em jogos, como número de peças e identificação de tabuleiros, para compor a função de avaliação; abordagem usada pelo Clune Player[3]. Vencedor das competições de 2007 e 2008, Cadia Player[4] usa técnicas de Monte Carlo para simular jogos e descobrir qual dos caminhos tem a maior chance de vitória.

O presente trabalho mostrará a construção de um jogador utilizando os métodos de Monte Carlo na função de avaliação, porém sem visar a competição, ficando restrito a jogos não adversaristas. O texto na seções seguintes está organizado da seguinte maneira:

- **Capítulo 2** revisa conceitos básico da Inteligência Artificial para jogos;
- **Capítulo 3** mostra a estrutura da linguagem utilizada para a descrição dos jogos em GGP;
- **Capítulo 4** apresenta como usar a linguagem de descrição para jogar;
- **Capítulo 5** revisa o método de simulações de Monte Carlo e mostra o algoritmo UCT.
- **Capítulo 6** encarrega-se de mostrar a implementação de um jogador;
- **Capítulo 7** será um breve comentário sobre os resultados obtidos;
- **Capítulo 8** conclui o trabalho.

CAPÍTULO 2

CONCEITOS BÁSICOS

A construção de um agente autônomo para jogos demanda a utilização de técnicas e conhecimentos diversos que estão além do escopo deste trabalho. Porém, algumas palavras chaves tangem todos os trabalhos nesta área. Este capítulo dará uma breve introdução a estes conceitos.

2.1 Árvore de Jogo

Um jogo pode ser representado como um grafo dirigido, uma *árvore*, representando o espaço de estados de um jogo. Em jogos determinísticos, cada *nó* na árvore representa um estado do jogo, e cada *aresta* representa um movimento. A *raíz* da árvore é o estado inicial do jogo, antes de qualquer jogador fazer qualquer movimento. Um *estado terminal* é a posição que as regras determinam que um jogo chegou ao fim.

Um nó é *expandido* quando todos os seus sucessores são descobertos da posição representada pelo nó. Um sucessor direto do nó é denominado *filho* do nó. O predecessor direto é chamado *pai*. A raiz, ou o estado inicial do jogo, é o único nó sem um pai. Da mesma forma, nós terminais são os únicos que não têm filhos.

A árvore do jogo é gerada expandindo todo nó, a partir do nó raiz, até todos o nós terminais serem atingidos. Cada nó terminal tem um valor associado, conhecido como *recompensa*, para cada jogador.

Examinar completamente a árvore de um jogo, da raíz até os terminais, é garantidamente a melhor estratégia para vencer um jogo. No entanto, para os jogos mais interessantes a árvore completa é extremanente grande, logo inviável computacionalmente.

2.2 Busca

Quando a árvore de jogo é muito grande para ser gerada integralmente, uma *árvore de busca* é usada em seu lugar. A árvore de busca é apenas parte da árvore de jogo. A raíz da árvore de busca representa o estado sob investigação. Esta árvore é gerada durante o processo de busca. Nós que ainda não foram expandidos são chamados *folhas*. Quando um nó terminal não é alcançado a recompensa associada a ele é dado por uma *função de avaliação* (ou *heurística*). A heurística produz uma recompensa estimada de quão bom seria para o jogador atingir aquele estado.

CAPÍTULO 3

GAME DESCRIPTION LANGUAGE

Para que a construção de um sistema autônomo em GGP seja possível, é preciso uma linguagem padrão para a descrição dos jogos. *Game Description Language*[7](GDL), derivada da lógica de Primeira Ordem e um subconjunto de *Knowledge Interchage Format*, é a linguagem usada em todas as competições realizadas até hoje. GDL é puramente axiomática, nenhuma algebra é incluida na linguagem. Caso o jogo necessite, as regras aritiméticas pertinentes devem ser incluidas na descrição.

A classe de jogos que pode ser descrita em GDL pode ser classificada como *n-jogadores* ($n \geq 1$), *determinística*, *perfeitamente informada* com *movimentos simultâneos*. "Determinística", exclui os jogos que contenham elementos aleatórios, como Gamão, onde a movimentação das peças é feita de acordo com um lançamento de dados. Já "perfeitamente informado"proíbe que qualquer informação sobre o estado atual seja escondida de algum jogador, o que é comum na maioria dos jogos de cartas. Finalmente "movimentos simultâneos"permite descrever jogos como Jan-ken-pon (pedra-papel-tesoura), onde todos os jogadores agem de uma vez, mas sem descartar jogos com alternância nas movimentações entre jogadores, como xadrez ou damas, restringindo todos os jogadores, exceto um, a executar apenas a ação "no-op". GDL também é *finita* em muitos aspectos: o espaço de estados consiste em muitos estados finitos; há também um número finito e fixo de jogadores; cada jogador tem um número finito de possibilidades de ações em cada estado e os jogos devem ser formulados de maneira que culmine em um estado terminal com um número finito de movimentos. Cada estado terminal está associado com um valor (*goal*) para cada jogador.

Os jogos em GDL são modelados como máquinas de estado, em que cada estado pode ser descrito como um conjunto de fatos verdadeiros em um determinado tempo. Um destes estados é projetado para ser o estado inicial. As transições são determinadas combinando

as ações de todos os jogadores. O jogo prossegue até que um dos estados finais seja atingido.

3.1 Estrutura da linguagem

Um pequeno conjunto de palavras chaves é usado para a descrição dos jogos: *role*, *init*, *next*, *true*, *does*, *terminal*, *goal* e *distintic*. Para ilustrar o uso na linguagem na descrição de um jogo, tomaremos como exemplo a descrição parcial do **jogo da velha**, onde os estados correspondem a configuração de preenchimento das células de uma matriz $3x3$ com x ou o. A descrição completa se encontra no apêndice.

3.1.1 Jogo da velha em GDL

Role

A palavra chave *role* é usada para descrever quais serão os jogadores da partida. Neste caso temos um jogador para marcar x e outro o. Vale notar a notação infixada que o GDL usa, diferente do Prolog que usa notação prefixada, onde a mesma declaração do jogador x seria feita como `role(xplayer)`.

```
(role xplayer)
(role oplayer)
```

Init

O estado inicial é constituído do conjunto de argumentos dos predicados *init*. O jogo da velha se inicia com todas as células em branco e o jogador x com o direito a jogada.

```
(init (cell 1 1 b))
(init (cell 1 2 b))
      ⋮
(init (cell 3 3 b))
(init (control xplayer))
```

True

O predicado *init* pode ser visto como um restrição da relação *true*, usada para escrever o que é verdade em um estado qualquer durante o decorrer do jogo. Em contrapartida, *init* só pode ser usado para a descrição do estado inicial. Após o primeiro movimento, a descrição do estado atual seria:

```
(true (cell 1 1 b))
      ⋮
(true (cell 2 2 x))
(true (control oplayer))
```

indicando que o jogador x marcou o centro da matriz e que agora o controle passa para o jogador o.

Next

Análoga à palavra chave *true*, o predicado *next* determina quais fatos serão verdades no próximo estado. A marcação do direito à jogada é feito da seguinte maneira:

```
(<= (next (control xplayer))
    (true (control oplayer)))
(<= (next (control oplayer))
    (true (control xplayer)))
```

O primeiro predicado *next* mostra que na próxima rodada o controle será do jogador x somente se no estado atual ele pertencer a seu oponente. A segunda declaração empreende o mesmo papel para indicar ao jogador o quando será sua vez.

Legal

As regras do jogo da velha permitem que um jogador marque uma célula apenas se ela ainda não foi marcada (contém um b) e seja sua vez de jogar. Para descrever estas regras,

a relação *legal*, juntamente com os demais predicados vistos, toma como parâmetro um jogador e ação a que ele tem direito. Em GDL, termos não unificados são marcados com o símbolo de interrogação.

```
(<= (legal ?player (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?player)))
(<= (legal xplayer noop)
    (true (control oplayer)))
(<= (legal oplayer noop)
    (true (control xplayer)))
```

A primeira relação descreve a regra fundamental do jogo da velha. As outras duas relações simulam a alternância entre o controle das jogadas, causando um dos jogadores a tomar a ação *noop*[1] quando não estiver no controle.

Distinct

Usado para distinguir duas proposições, o predicado *distinct* será verdadeiro somente se seus argumentos não forem iguais.

```
(<= (next (cell ?x ?y b))
    (does ?player (mark ?m ?n))
    (true (cell ?x ?y b))
    (distinctCell ?x ?y ?m ?n))
(<= (distinctCell ?x ?y ?m ?n)
    (distinct ?x ?m))
(<= (distinctCell ?x ?y ?m ?n)
    (distinct ?y ?n))
```

[1]No Operation, *nenhuma ação*

Terminal

Para determinar se o estado atingido é um estado final, os predicados *terminal* devem ser consultados.

```
(<= terminal
    (line x))
(<= terminal
    (line o))
(<= terminal
    (not open))
```

Destas regras tem se que o jogo acaba ao marcar uma sequência de três células, para ambos os jogadores, ou quando não há mais espaços em brancos.

Goal

Ao final do jogo, cada jogador terá uma pontuação de acordo com o seu desempenho.

```
(<= (goal xplayer 100)
    (line x))
(<= (goal xplayer 50)
    (not (line x))
    (not (line o))
    (not open))
(<= (goal xplayer 0)
    (line o))
```

Este recorte da descrição revela a pontuação do jogador x caso ele vença, empate ou perca, ganhando 100, 50 ou 0, respectivamente. As mesmas regras para o jogador o também são encontradas na descrição.

3.2 Considerações

Algumas consideraçãoes a cerca da linguagem e suas propriedades devem ser feitas.

3.2.1 Predicados auxiliáres

As relações fundamentais vistas acima contam com o uso de predicados auxiliáres na descrição. Usado na relação *init*, o predicado *(cell ?x ?y ?p)* indica qual marca a célula *xy* contém. O número de regras auxiliáres que podem ser usadas na descrição é indefinido. Esta descrição do jogo da velha contém outros exemplos, como: *control*, *distinctCell*, *line*, *open* e outros.

3.2.2 Nenhuma dica léxica

Um agente GGP não pode contar com a descrição do jogo para construír uma função heurística. Exceto pelas palavras chave, nenhum outro conjunto de símbolos tem algum significado para o jogador. Na verdade, durante a competição, os símbolos usados na descrição são embaralhados para previnir que o jogador retire alguma dica léxica. Por exemplo, uma das estruturas mais básicas dos jogos é o tabuleiro. Na competição a mesma descrição do tabuleiro poderia ser algo como:

```
(init (homer bart bart santa))
(init (homer bart lisa santa))
        ⋮
(init (homer magie magie santa))
```

3.2.3 Problema do Quadro (The Frame Problem)

Segundo Ginsberg [5], é um questionamento de como representamos o fato de que as coisas tendem a continuar as mesmas na ausência de informações contrárias. Em GDL, o próximo estado sempre é definido pela relação *next*, logo a regra para o problema do quadro é declarar explicitamente o que será verdadeiro após as transição de estados. No

exemplo do predicado *distinct*, as regras declaram que uma célula permanecerá em branco caso nenhum jogador a tenha marcado na rodada atual.

3.2.4 Jogos Finitos

Como mencionado anteriormente, somente jogos de tamanho finito são permitidos. Enquanto jogos como jogo da velha são garantidamente finitos, a maioria dos jogos, como o mundo dos blocos, tem ciclos em suas máquinas de estados que permitem uma infinita sequência de ações (como por exemplo, empilhar e desempelhar o mesmo bloco idefinidamente). Portanto o construtor da descrição de garantir que o mesmo estado não seja atingido duas vezes na mesma partida, e que todas as partidas invariavelmente atinjam um estado final. A maneira mais comum para fazer isso é adicionar a descrição um contador (*step*), incrementar o seu valor a cada ação e acabar o jogo quando um certo número máximo for atingido.

CAPÍTULO 4

PROVA AUTOMÁTICA DE TEOREMAS

Para processar as descrições em GDL, é imprescindível o uso de um Provador Automático de Teoremas (Automatic Theorem Proving, *ATP*), que é um programa para provar teoremas matemáticos. A linguagem em que os teoremas são escritos é logica, comumente lógica de primeira ordem. Teoremas são compostos de axiomas e hipóteses que levam a uma conclusão (ou não). Sistemas ATP produzem provas que descrevem como e porquê a conclusão é uma consequência lógica dos axiomas e das hipóteses.

As provas produzidas por um sistema ATP são construídas para serem facilmente compreensíveis. Por exemplo, se o jogo Torres de Hanoi fosse formulado como um teorema, a prova descreveria a sequência de movimentos necessários para resolver o problema.

4.1 Inferência

Em lógica, uma *regra de inferência* é uma regra de raciocínio sobre um conjunto de sentenças, chamado *premissas*, e um segundo conjunto de sentenças chamado *conclusão*. Abaixo uma regra de inferência chamada Modus Pones.

1. `A` $\Rightarrow$ `B` (se A é verdade então B é verdade)
2. `A` (A é verdade)
3. `B` , por *1* e *2* (B é verdade)

Para provar um teorema, os axiomas da teoria a ser provada são primeiramente colocados em um forma padrão chamada forma clausal. Um algoritmo de inferência é então aplicado exaustivamente para o resultante conjunto de clausulas na busca por uma contradição. O componente principal que realiza o algoritmo de inferência é conhecido como *máquina de inferência*.

As duas seções seguintes descrevem como um sistema ATP é usado para retirar informações cruciais para um jogador compreender a estrutura lógica de um jogo e realizar

movimentos permitidos pelo mesmo.

4.2 Determinando jogadas legais

A partir do estado inicial, todos os predecessores são derivados usando as regras *legal* e *next*, como pode ser visto no exemplo abaixo.

Os axiomas abaixo foram retirados da descrição do estado inicial (regras *init*) do jogo da velha.

1. `cell(1 1 b)`
2. `control(xplayer)`

Para sumarizar as jogadas legais em um determinado estado do jogo, o predicado *legal* deve ser usado sobre a base de fatos que constitui o estado atual.

3.
```
(<= (legal ?player (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?player)))
```

Por se tratar de uma linguagem deveriada da lógica de primeira ordem, as descrições em GDL podem ser transcritas para outras linguagem de mesma ascenção, como Prolog. O predicado *legal* acima, por exemplo:

3. `(cell(X, Y, b)` $\wedge$ `control(Player))` $\Rightarrow$ `legal(Player, (mark(X, Y)))`

As jogadas possíveis em um determinado estado são inferidas da base de conhecimento aplicando-se as regras de movimentação.

4. `legal(xplayer, (mark (1, 1)))`, por *1, 2* e *3*

4.3 Constituindo um novo estado

Os estados subsequêntes ao inicial, que é descrito pelo conjunto *init*, são determinados após os jogadores realizarem seus movimentos, através da declaração *does* que deve ser inserida na base de fatos. Seguindo o exemplo anterior, caso o jogador x escolha a ação de marcar a célula (1, 1), o predicado *does* referente a esta jogada será:

5. `(does (xplayer (mark ?x ?y))))`

Em poder das ações tomadas por todos os jogadores na rodada atual, o próximo estado pode ser encontrado com auxílio do predicado abaixo. É importare relembrar que em todas as rodadas todos os jogadores realizam uma ação. Mesmo nos jogos onde há alternância de turnos, como o próprio jogo da velha, o oponente fica restrito a ação *noop*.

6.
```
(<= (next(cell(?x ?y x)))
    (does (xplayer (mark ?x ?y))))
```

Assim o primeiro termo constituinte do novo estado é inferido. Com a aplicação das demais regras *next* tem-se um estado completo.

7. `next(cell(1 1 x))`, por *5* e *6*

CAPÍTULO 5

HEURÍSTICA

O modo mais comum para aplicar Inteligência Artificial à jogos é utilizar métodos como *A** e *MiniMax*, para representar a árvore de busca, em conjunto com uma heurística, desenvolvida com algum conhecimento prévio do jogo. No entanto, desenvolver uma heurística em GGP não é uma tarefa trivial. A maioria dos jogadores GGP tem soluções únicas para esse problema, pois não existe nenhuma pesquisa sólida que diga como gerar uma boa heurística generalista. Algumas abordagem usam características dos jogos, como número de peças, que podem contribuir com a heurística em muitos jogos, mas que em outro conjunto de jogos pode não funcionar ou até mesmo causar prejuízo na avaliação.

Ao lidar com GGP é portanto mais interessante investigar funções de avaliação que não dependem de características específicas, que podem não existir ou fazer sentido em todos jogos. Contar peças, por exemplo, pode ser bom em xadrez mas não faz sentido algum no jogo da velha. A solução adota neste trabalho é uma função de avaliação que explore os estados e descubra através de simulações quão bom ou ruim eles são.

5.1 Monte Carlo

O conjunto de métodos de Monte Carlo[10] necessita apenas de experiência na interação com o sistema para estimar o valor de uma função de avaliação. A partir da descrição da mecánica do sistema é possível obter a experiência simulando as interações. Um conjunto de simulações realizadas com um número de passos finitos é conhecida como *tarefa episódica*. Uma única simulação da tarefa é chamada de *episódio* e a pontuação obtida durante o episódio é o *retorno*.

A ideia fundamental é aprender com a média dos retornos obtidos durante as simulações. A experiência é divida em episódios que eventualmente levam a um objetivo e no final deste as estimativas de valores são obtidas. Os métodos de Monte Carlo são,

de certa forma, incrementais, pois as médias são alteradas após cada episódio. O termo "Monte Carlo"é frequentemente usado em qualquer estimativa que envolva algum componente randômico. Neste caso, é usada aleatoriedade para percorrer os estados durante as simulações.

Cada estado mantem uma estimativa da recompensa total acumulada, $V(s)$, que o jogador irá atingir caso vá para o estado s. Ao final da simulação de cada episódio, o retorno é propagado retroativamente para todos os estados visitados no episódio e a média é recalculada. A média dos retornos pode ser calculada incrementalmente da seguinte maneira:

$$V(s) \leftarrow V(s) + \frac{1}{n(s)}[R - V(s)] \tag{5.1}$$

onde $n(s)$ é o número de visitas ao estado s e R o retorno obtido no término da simulação.

5.2 UCT

Monte Carlo basea-se em repetidas simulações randômicas para colher resultados. A estratégia mais simples é repetir as simulações até esgotar-se o tempo, e escolher o movimento com melhor retorno. Usar esta estratégia no entanto, faz que o tempo seja gasto igualmente na exploração de bons e de maus movimentos. Se em vez disso fosse usada a informação já obtida, deixando os bons movimentos com um peso maior, o desempenho do jogador seria melhorado, já que usar o tempo explorando quão ruim um mau movimento é, é perda de tempo.

Este problema é uma variante do *multi armed bandit*[1], onde uma máquina de caça-níquel tem múltiplas alavancas. Cada alavanca produz uma recompensa aleatória de uma distribuição desconhecida, e a distribuição de recompensas de uma alavanca pode ser diferente das outras. O objetivo é maximar a coleta de recompensa após iterações consecutivas. Puxar alavancas diferentes pode ensinar mais sobre cada uma, porém recompensas maiores podem ser perdidas por usar uma alavanca que não é ótima.

Existem diferentes abordagens para este problema. Uma das estratégias conhecidas é

[1] http://en.wikipedia.org/wiki/Multi-armed_bandit

o algoritmo UCB[1]. UCB está para Upper Confidence Bounds, onde o algoritmo garante um limite superior (upper bound) do arrependimento causado por não puxar a alavanca ótima. A ideia é que cada alavanca possua um registro da média das recompensas atingidas ao puxa-lá e um viés[2]. O elemento chave desta estratégia está em como o viés é calculado. No algoritmo UCB1 ele é dado pela fórmula:

$$\sqrt{\frac{2 \ln n}{n_j}} \tag{5.2}$$

onde n_j é o número de vezes que a alavanca j foi usada e n o total de jogadas feitas.

Ao aplicar UCB a jogos, o cenário sofre algumas alterações. No lugar de uma única máquina caça-níquel com alavancas independentes, cada alavanca da primeira máquina vai gerar uma outra máquina ou produzir uma recompensa. Isso corresponde a realizar um movimento no jogo e atingir novo estado ou uma recompensa de um estado final. Ainda sim é possível usar o algoritmo UCB para resolver o problema. O novo método é chamado UCT[6], que significa UCB aplicado a árvores (UCB applied to Trees).

O algoritmo funciona como a estratégia de simualção de Monte Carlo, mas no lugar de escolher ações randômicas ele usa UCB em cada estado para explorar a recompensa que a ação proporcionará. Utilizando a média dos retornos, a fórmula para UCT é:

$$V(s) + C_p \sqrt{\frac{\ln N(s)}{N(s,a)}} \tag{5.3}$$

A constante C_p indica quanto o bônus será levado em consideração na escolha. $N(s)$ é o total de visitas ao estado s durante as simulações e $N(s,a)$ o número de vezes em que a ação a foi tomada.

A grande vantagem do UCT é que ele não precisa de nenhuma heurística para dar um bom resultado, já que usa recompensas reais para estimar o valor dos movimentos. Também foi provado matematicamente que a probabilidade de escolher uma ação ótima converge para 1 quando o número de simulações cresce.

[2]Diferença entre o valor esperado do estimador e o verdadeiro valor do parâmetro a estimar. http://pt.wikipedia.org/wiki/Viés

Infelizmente existem algumas desvantagens. Se a árvore do jogo for muito grande ou cada mudança de estado for computacionamente complexa para ser calculada, o algoritmo pode apenas algumas vezes ou nunca atingir um estado terminal, deixando a tomada de decisão infundada. Há também o caso em que um movimento inicialmente parace bom mas na verdade é mau, podendo enganar o algoritmo se ele não tiver tempo o suficiente para simular e consertar o erro.

CAPÍTULO 6

IMPLEMENTAÇÃO

A construção do jogador foi dividada da seguinte maneira: um módulo para fazer a análise da descrição em GDL, um provador de teoremas em Prolog e a parte responsável pela inteligência do jogador, implementada usando UCT. A maior parte do código foi escrita em *Ruby*, porém há um resquício de código em *C* e o YAP foi escolhido como o provador de teoremas Prolog.

6.1 GDL Parser

As descrições dos jogos em GGP são escritas em GDL, que utiliza o padrão Knowledge Interchange Format (KIF). No entanto, como o provador de teoremas foi escrito em Prolog é necessário uma tradução para um formato que o Prolog entenda. Como GDL é um subconjunto da lógica de primeira ordem, essa tarefa é facilmente cumprida. Por exemplo a regra abaixo do jogo da velha:

```
(<= (legal ?player (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?player)))
```

será traduzida para

```
legal(Player, mark(X, Y)):-
    cell(X, Y, b),
    control(Player).
```

O predicado *distinct*, no entanto, necessita de um tratamento especial. Ele recebe duas cláusulas como parâmetro e representam o relacionamento de *não igualdade* entre elas. Em Prolog:

```
distintic(X, Y):- X \= Y.
```

Da mesma maneira, o predicado *or* existe para a relação *ou lógico* entre os parâmetros. Os primeiros jogos utilizavam esta relação com um número arbitrário de parâmetros, mas isto acabou sendo abandonado. Para manter a compatibilidade com os jogos anteriores, foi incluido o código abaixo:

```
or(X, Y):-
    call(X) ; call(Y).
or(X, Y, Z):-
    call(X) ; call(Y) ; call(Z).
or(X, Y, Z, W):-
    call(X) ; call(Y) ; call(Z) ; call(W).
```

Para o parser da descrição, foi utilizada a biblioteca *Rex*, para análise léxica, e *Racc*, para a sintática. A dupla utilizada é equivalente ao *Flex* e *Bison*, porém com implementação para Ruby.

6.2 Provador de Teoremas

O provador de teoremas é iniciado assim que análise da desscrição do jogo termina. Todos os predicados do jogo são inseridos na base, exceto os *inits*, que serão enviados posteriormente para compor o estado inicial. Utilizando o YAP como interface, ele é usado para calcular: os movimentos legais de um jogador em um determinado estado; o próximo estado, dado o estado atual e uma ação válida; a recompensa alcançada ao atingir um estado (*goal*) e se um determinado estado é terminal ou não.

Após a análise da descrição, o arquivo `game_description.yap`, contendo os predicados declarados pelo jogo, é gerado e posteriormente carregado no YAP utilizando a cláusula `consult`.

A comunicação entre o YAP e o jogador é feita utilizando *Unix Sockets*. Uma primeira versão do jogador foi implementada com *TCP Sockets*, porém o *buffer* comum em conexões TCP, deixava inviável o uso para a construção do jogador. Como os movimentos legais precisam ser calculados para se obter o estado seguinte, o buffer atrasava a saida e, consequentemênte, o desempenho do jogador.

Para realizar qualquer ação no provador, o estado atual é enviado e inserido na base de fatos com o predicado `assert`. Após a avaliação da cláusula desejada (*next*, por exemplo), os fatos são retirados pelo predicado `retract`. Isto deixa a comunicação ligeiramente mais lenta pois inserção e remoção de fatos em tempo de execução é um processo custoso.

6.3 UCT Player

A escolha das ações realizadas pelo jogador é feita pelo algoritmo UCT. Como descrito antes, este algoritmo não precisa de uma heurística, no entanto há uma fase de análise na competição que é usada para fazer o maior número possível de simulações, para se obter uma média confiável dos bonus. Caso não haja tempo hábil para encontrar ao menos uma solução com retorno maior que zero, é provável que ela não seja encontrada durante o jogo. A implementação deste algoritmo é bem direta:

Algorithm 6.3.1: UCTSimulate(*state*)

while HAS_TIME

do
- **while not** $TERMINAL(state)$
  - **do**
    - $VISIT(state)$
    - $legals \leftarrow LEGALS(state)$
    - $unexplored \leftarrow UNEXPLORED(legals)$
    - **if** *unexplored*
      - **then** $UCTSimulate(unexplored)$
      - **else**
        - $next\_state \leftarrow BEST\_BONUS(state)$
        - $UCTSimulate(next\_state)$

A constante da formúla $C_p\sqrt{\frac{2\ln n}{n_j}}$ tem que ser empiricamente ajustada. Testes mostraram

que o valor 40 é razoável para a maioria dos jogos. Escolher um valor 0 (ignorar totalmente o bonus) ou 100, pode fazer a performance do jogador cair drasticamente.

Como no inicio do jogo não há nenhum movimento explorado, o algoritmo será totalmente randômico. Depois de algumas simulações, as diferentes recompensas do estado inicial serão levadas em consideração e os movimentos seguintes novamente aleatórios. Com a exploração de um número maior de ações, o algoritmo passará a ser mais determinístico para a escolha do próximo estado.

CAPÍTULO 7

RESULTADOS

Apesar de usar uma ótima solução para GGP, dado os resultados das competições de 2007 e 2008, onde o ganhador foi implementado utilizando UCT, a implementação deste trabalho não obteve os resultados esperados.

Em grande parte do desenvolvimento o jogo utilizado para testes foi o *8 puzzle*, onde um tabuleiro $3x3$, contendo números de 1 a 8 fora de sequência e uma célula vazia, deve ser ordenado. Neste jogo não houve sequer uma vez em que a recompensa obtida durante as simulações fosse diferente de zero. Mesmo aumentanto do tempo de análise pré jogo para 10 minutos, não houve alteração de comportamento. Desnecessário um gráfico para tal resultado.

Entretanto, para o jogo *Torres de Hanoi*, com 5 discos, o resultado colhido foi ligeiramente animador. Em todas as simulações, utilizando um tempo de análise de 15 segundos, a pontuação final obtida foi 60 pontos, das possíveis entre 0-40-60-80-100. Novamente, aumentou-se o tempo de análise e o resultado obtido foi o mesmo.

CAPÍTULO 8

CONCLUSÃO

O novo ramo da Inteligência Artificial, *General Game Playing*, ainda não conta com técnicas consolidadas para a resolução dos problemas propostos. O que torna uma área excelente para desenvolvimento de novos métodos, como extração de características comuns em jogos, ou aplicação de métodos usados comumente em outras áreas, como Monte Carlo.

Neste trabalho foi apresentado a construção de um jogador GGP utilizando as técnicas de Monte Carlo e UCT. Partindo da linguagem utilizada na descrição dos jogos, a mecânica de como derivar as informações também foi exposta.

No Capítulo 5 foi mostrado que o método de Monte Carlo por si só não é suficiênte para a construção do jogador. O aprimoramento com Upper Confidence Bonus aumenta a taxa de sucesso enormemente.

Os detalhes da implementação de um jogador, utilizando os conceitos apresentados nos capítulos anteriores, foram mostrados no Capítulo seguinte. Peculiaridades da linguagem GDL devem ser levadas em consideração na transcrição para Prolog. A comunicação entre o provador de teoremas e o jogador é um fator muito importante no desempenho. Já a implementação do algoritmo UCT é praticamente direta, variando-se apenas a constante de utilização do bônus.

No entanto, como exposto no resultados, ainda há muito o que progredir. Na implementação alternativas e melhoramentos na seleção das ações devem ser incorporados ao jogador para aumentar a taxa de sucesso.

BIBLIOGRAFIA

[1] Peter Auer, Nicolò Cesa-Bianchi, e Paul Fischer. Finite-time analysis of the multi-armed bandit problem. *Machine Learning*, 47(2):235–256, May de 2002.

[2] Murray Campbell, A. Joseph Hoane, Jr., e Feng-hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[3] James Edmond Clune, III. *Heuristic evaluation functions for general game playing.* Tese de Doutorado, Los Angeles, CA, USA, 2008. Adviser-Korf, Richard E.

[4] Hilmar Finnsson e Yngvi Björnsson. Simulation-based approach to general game playing. *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*, páginas 259–264. AAAI Press, 2008.

[5] Matt Ginsberg. *Essentials of artificial intelligence.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[6] Levente Kocsis e Csaba Szepesvari. *Bandit based Mont-Carlo Planning.* Springer, 2006.

[7] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, e Michael Genesereth. *General Game Playing Specification*, 2008.

[8] Jonathan Schaeffer, Robert Lake, Paul Lu, e Martin Bryant. Chinook the world man-machine checkers champion. *AI Magazine*, 17(1), 1996.

[9] Stephan Schiffel e Michael Thielscher. Fluxplayer: a successful general game player. *AAAI'07: Proceedings of the 22nd national conference on Artificial intelligence*, páginas 1191–1196. AAAI Press, 2007.

[10] Richard S. Sutton e Andrew G. Barto. Reinforcement learning i: Introduction, 1998.