FEUP

Inteligência Artificial

2007/2008 - 2S

Resolução do Solitário

Same Game

27 de Maio de 2008

Ivo Paz dos Reis

ei05021@fe.up.pt

Sérgio Miguel Fontes de Vasconcelos

ei05074@fe.up.pt

Índice

1. Introdução

Este projecto visa implementar o jogo SameGame com metodologias que simulem Inteligência Artificial a nível de jogadas.

O jogo em si foi originalmente criado por Kuniaki Moribe em 1985 com o nome de *ChainShot!* Rapidamente começaram a surgir novas versões do jogo, sendo que actualmente o SameGame também é conhecido por Bobblet, JawBreaker, BubbleBreaker, etc.

A vertente de inteligência artificial não se encontra em nenhuma das versões actualmente no mercado, o que é uma mais valia para este trabalho.

1.1 Enquadramento

Este trabalho insere-se no contexto da disciplina de Inteligência Artificial do 3º ano do Mestrado Integrado em Engenharia Informática e Computação.

O tema deste trabalho enquadra-se na temática da pesquisa sistemática, mais propriamente na determinação de soluções do puzzle SameGame, utilizando variados métodos de pesquisa aliados a diferentes heurísticas.

1.2 Motivação

O maior atractivo à realização deste trabalho consiste na possibilidade de, através da implementação de diferentes heurísticas, criar algoritmos de pesquisa que sejam suficientemente eficazes para superar a prestação de um jogador humano.

Inicialmente meramente académico, o interesse tornou-se mais forte ao testar as implementações já existentes, que não oferecem funcionalidades relacionadas com inteligência artificial.

1.3 Objectivos

A implementação deste jogo tem como principal objectivo estudar as diferentes estratégias adoptadas pelos jogadores humanos de forma a obterem a melhor pontuação possível. Partindo deste estudo, avalia-se se a criação de diferentes heurísticas que se aproximem destes comportamentos, aplicadas a diferentes métodos de pesquisa, resultam efectivamente em resultados que se aproximem das soluções óptimas.

2. Funcionalidades

Neste jogo estão implementadas variadas funcionalidades, tanto ao nível de jogabilidade, como ao nível da inteligência artificial.

2.1 Configurar número de linhas, colunas e cores

Acedendo ao submenu Opções do menu Jogo, é possível configurar as dimensões do tabuleiro de jogo, bem como o número de cores das bolhas. Desta forma torna-se fácil criar puzzles de diferentes dificuldades. É também possível definir os limites para as pesquisas DLS e IDDFS explicadas posteriormente, bem como definir o tempo de execução da jogada do computador em milissegundos.

2.2 Recuar jogadas (*undo*)

De forma aumentar a jogabilidade, encontra-se no menu Jogadas a opção de recuar para a última efectuada. Recorrendo a uma pilha, são armazenados todos os estados passados de jogo. Como tal, é possível utilizar esta opção para recuar no jogo até ao estado inicial.

2.3 Avançar jogadas (*redo*)

Esta opção insere-se, tal como a anterior, no interesse existente em oferecer uma boa jogabilidade. Igualmente no menu de Jogadas, esta opção permite ao jogador desfazer um recuo efectuado no decorrer do jogo. Pode assim avançar novamente para a última jogada efectuada. Utiliza também uma pilha, que armazena o registo de todas as jogadas que foram recuadas.

2.4 Sugestão de jogada

O utilizador pode requisitar a sugestão, por parte do computador, da jogada mais promissora para o estado corrente de jogo. Esta sugestão varia consoante o nível de inteligência artificial que estiver configurado.

2.5 Competir contra o computador

Esta é uma das funcionalidades mais interessantes implementadas neste jogo, pois oferece ao utilizador o desafio de tentar superar a prestação do computador no que diz respeito à resolução de um puzzle. É gerado um puzzle aleatório com as características previamente configuradas, isto é, número de linhas, colunas e cores. O puzzle é inicialmente resolvido pelo computador, sendo mostrada uma mensagem com a pontuação obtida por este, bem como o número de jogadas de que necessitou, e o tempo gasto. De seguida este mesmo puzzle é carregado, e cabe ao jogador tentar obter uma pontuação superior à obtida pelo computador. Existem diferentes níveis de dificuldades, que são também previamente configurados.

2.6 Cálculo de solução completa de puzzles

Esta funcionalidade permite ao jogador observar, em tempo real, o conjunto de jogadas calculadas pelo computador para a resolução de um determinado puzzle. Depois de calculada a solução, é criada uma animação, com um intervalo entre jogadas configurável. Desta forma é criada a sensação de que o computador está a jogar em tempo real, simulando o comportamento humano.

2.7 Configurar nível de inteligência artificial

A configuração do nível de inteligência artificial consiste em aliar uma heurística a um método de pesquisa. Visto que foram implementados vários métodos de pesquisa e várias heurísticas, é possível criar bastantes combinações de pesquisa informada a ser computada.

3. Estrutura do Programa

A estrutura do programa pode ser discriminada, sobre uma perspectiva de alto nível, em três módulos principais:

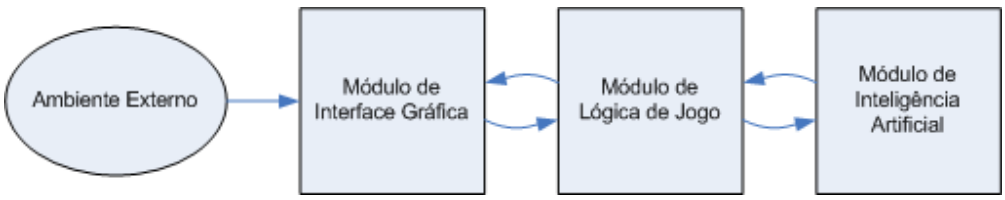

Figura 1 - Diagrama da Estrutura do Programa

3.1 Módulo de Interface Gráfica

Este módulo é responsável pela representação gráfica do estado do jogo. Incorpora também os diversos menus, permitindo assim a interacção com o utilizador.

Através dos comandos do utilizador são activados eventos responsáveis pela ligação com o módulo de lógica do jogo. Sendo assim, existe uma ligação bilateral entre estes módulos. A interface envia comandos para o motor do jogo, que após o seu processamento, devolve o novo estado de jogo ou os dados necessários para a representação de informação relevante, tal como a pontuação ou finalização.

3.2 Módulo de Lógica de Jogo

O núcleo da implementação do Samegame encontra-se neste módulo. São verificadas todas as condições/restrições inerentes ao correcto desenrolar do jogo. A pedido do utilizador, poderá ser invocada a funcionalidade do programa encontrar uma solução, sendo então encaminhado um pedido ao modo de inteligência artificial, especificando o modo e os parâmetros de pesquisa.

3.3 Módulo de Inteligência Artificial

Este módulo contém o conjunto dos algoritmos que permitem a pesquisa de soluções para os puzzles, que serão explicados mais pormenorizadamente no 4º Capítulo deste relatório .

4. Esquemas de Representação de Conhecimento

A informação necessária de representação do jogo é o conjunto de bolhas presentes no tabuleiro. Assim, existem várias bolhas com diferentes cores.

4.1 Implementação do Esquemas de Representação de Conhecimento

Para o armazenamento das bolhas, optou-se por utilizar um array nativo bidimensional, sendo cada um dos sub-arrays correspondente a uma coluna da matriz de bolhas. Esta escolha justifica-se por ser a estrutura de dados mais leve e de fácil manutenção, contendo as propriedades necessárias à implementação efectuada.

Cada bolha corresponde a um inteiro que representa a sua cor. Existem portanto diferentes identificadores numéricos para os diferentes tipos de bolhas.

Descrição do Motor do Jogo:

collectBubbles

Este é o algoritmo responsável pela recolha das bolhas a serem eliminadas, após uma jogada efectuada pelo utilizador. Seguindo as regras tradicionais do jogo, definimos que uma jogada é válida sempre que é efectuado um *click* em cima de uma bolha que tem imediatamente na sua área circundante (cima, baixo, esquerda ou direita), uma ou mais bolhas da mesma cor. Portanto, toda a cadeia de bolhas da mesma cor é eliminada até que sejam encontradas bolhas de cor diferente. Visto que as bolhas são geradas de forma aleatória, foi necessário escrever um algoritmo recursivo, que para cada bolha, recolhesse um conjunto de condições necessárias à decisão de remoção da mesma. O algoritmo collectBubbles utiliza um ArrayList de instâncias da classe Coordenada, que é devolvido pela função “vizinhos”, e que por sua vez recolhe as seguintes informações acerca de cada um dos vizinhos da bolha que está a ser analisada: verifica se é uma bolha válida (se existe no tabuleiro), e efectua a comparação de cores. Se ambas as condições se verificarem, armazena a coordenada do vizinho no ArrayList de vizinhos. Se nenhum dos vizinhos tiver a mesma cor, o procedimento collectBubbles termina de imediato. Caso

contrário, adiciona a bolha que foi clicada noutro ArrayList de Coordenada (selectedBubbles), que é um atributo da classe SameGame, e que por sua vez armazena todas as bolhas a eliminar, em cada jogada.

Finalmente, o ArrayList de vizinhos é percorrido, e se a coordenada do vizinho ainda não estiver contido em selectedBubbles (verificação feita pela função de retorno boleano contemBubble), é efectuada uma nova chamada do método collectBubbles, que por sua vez irá efectuar o mesmo conjunto de operações para a bolha em questão. A função contemBubble foi criada com o intuito de evitar chamadas repetidas do método collectBubbles, que devido às características da disposição das bolhas, poderia ter de recalcular inúmeras vezes informação calculada previamente. Esta função percorre o ArrayList selectedBubbles da classe.

- *deleteBubbles*

Esta função é responsável pela destruição das bubbles da matriz corrente, percorrendo o vector de coordenadas selectedBubbles, onde estão armazenadas as coordenadas de todas as Bubbles a ser eliminadas. Esta eliminação é conseguida colocando o valor null na matriz.

- *fallingBubbles e pushColumns*

Funções responsáveis pela reorganização do tabuleiro de jogo. Sendo a pushColumns responsável pela verificação da existência de algum array vazio na matriz de bubbles, deslocando todas as bubbles para a direita, e a FallingBubbles verificando se o valor da matriz é null, deslocando as bubbles para baixo.

- *vizinhos*

Esta função devolve um array de coordenadas de Bubbles, onde para uma dada bolha verifica as adjacentes e caso partilhem a mesma cor coloca-as nesse array.

5. Inteligência Artificial

Neste trabalho tentou-se criar, através de diversos algoritmos e heurísticas, a ilusão de que o computador é capaz de jogar de forma inteligente o jogo implementado. Esta "inteligência artificial" conseguiu atingir resultados satisfatórios, tendo sido modelada tendo em conta as melhores estratégias usadas por humanos. Nas secções seguintes serão explicados mais detalhadamente os métodos de pesquisa e as heurísticas usadas.

5.1 Métodos de Pesquisa

Todos os métodos de pesquisa usados constroem uma árvore cujos nós representam tabuleiros e as ramificações as jogadas efectuadas.

BFS – Breadth-First Search (Pesquisa Primeiro em Largura)

Este método de pesquisa não informado parte de um nó raiz e explora todos os seus filhos, para cada um dos filhos repete o processo até encontrar uma solução. É um método de pesquisa completo visto que, caso exista solução esta é sempre encontrada.

No entanto, devido a ser necessário guardar em memória todos os nós por visitar de cada ramificação da árvore, que no caso do Samegame podem atingir um número bastante elevado.

O factor de complexidade espacial deste algoritmo é de $O(b^d)$, sendo b o factor de ramificação (breadth) e d a profundidade da árvore (depth). Como cada nó é visitado e expandido pode considerar-se que a complexidade temporal é também de $O(b^d)$.

Para tabuleiros pequenos ou com poucas jogadas possíveis este algoritmo consegue encontrar a solução com menos jogadas, no entanto, como o custo espacial é exponencial com a ramificação da árvore, para tabuleiros maiores torna-se impossível encontrar a solução devido à falta de memória para armazenar todas as ramificações.

A estrutura de dados usada para este algoritmo foi uma fila (Queue) do tipo FIFI (First In First Out), visto que ao se expandir os filhos de cada nó estes são armazenados sequencialmente. Os nós visitados primeiro são os irmãos do último nó visitado até não restarem mais irmãos. Seguidamente são visitados os filhos do primeiro nó visitado.

Esquematização do algoritmo:

1. Coloca a raiz no início da fila
2. Retira o primeiro nó da fila e explora-o
   - Testa se é atingido o GameOver e termina
   - Senão cria todos os filhos deste nó e adiciona-os ao final da fila e repete a partir de 2.

DFS – Depth-First Search (Pesquisa Primeiro em Profundidade)

Este método de pesquisa não informado parte de um nó raiz e explora o primeiro filho, expandindo depois o primeiro filho dele indo tão longe quanto for possível Quando não haver mais ramificações o algoritmo volta atrás até ao nó mais recente por visitar.

Para o caso do SameGame este algoritmo é bastante rápido pois encontra sempre uma solução independentemente do ramo que escolher, pois é sempre possível terminar o jogo. Isto só não se verifica impondo a restrição de tabuleiro vazio, onde se obriga o algoritmo a voltar atrás caso chegue a uma solução não válida.

O factor de complexidade espacial deste algoritmo, comparativamente ao BFS, é muito reduzido, pois não visita todos os nós de cada nível antes de passar para o nível seguinte. A sua complexidade espacial é portanto O(h), sendo h a altura (height) do ramo maior. À semelhança do BFS, a complexidade temporal é também $O(b^d)$.

Tal como descrito acima este algoritmo é capaz de encontrar uma solução muito rapidamente, tendo em conta a especificidade do problema do Samegame, que permite terminar sempre o jogo independentemente das jogadas que se fizer. Mesmo para tabuleiros titânicos este algoritmo consegue encontrar soluções.

A estrutura de dados usada para este algoritmo foi uma pilha (Stack) do tipo LIFO (Last In First Out), pois sempre que se visita um novo nó os filhos dele terão prioridade em relação aos restantes nós por visitar.

Esquematização do algoritmo:

1. Coloca a raiz no início da pilha
2. Retira o primeiro nó da pilha e explora-o
    - Testa se é atingido o GameOver e termina
    - Senão cria todos os filhos deste nó e adiciona-os ao início da pilha e repete a partir de 2.

DLS – Depth-Limited Search (Pesquisa em Profundidade Limitada)

A pesquisa em profundidade limitada é essencialmente a pesquisa em profundidade, mas que quando se atinge um limite predefinido deixa de se criar os filhos do nó que atingiu essa profundidade, passando para o próximo nó a visitar.

Este algoritmo é bastante eficiente para problemas em que o nível da profundidade da árvore é relevante, o que não é o caso do SameGame. Se a profundidade definida for baixa corre-se o risco de não encontrar nenhuma solução. No entanto, este método provou ser relativamente rápido, principalmente quando comparado com o BFS.

O factor de complexidade espacial e temporal deste algoritmo é equivalente ao do DFS, ou seja, O(h) e $O(b^d)$ respectivamente.

A estrutura de dados deste método é a mesma que a do DFS.

Esquematização do algoritmo:

1. Coloca a raiz no início da pilha
2. Enquanto Limite não for atingido
    - Retira o primeiro nó da pilha e explora-o
        - Testa se é atingido o GameOver e termina
        - Senão cria todos os filhos deste nó e adiciona-os ao início da pilha e repete a partir de 2.

- Descarta nó corrente e avança para o próximo nó não visitado, repetindo a partir de 2.

IDDFS – Iterative DeepeningDepth-first Search (Pesquisa em Profundidade Iterativa)

A pesquisa em profundidade iterativa é uma sucessão de pesquisas em profundidade limitada, em que o limite da pesquisa começa num valor baixo e vai aumentando até se encontrar uma solução. A cada aumento do limite da profundidade é descartada a árvore criada até então, o que pode parecer à primeira vista ineficiente, mas na verdade este método expande apenas mais 11% de nós do que o BFS ou o DFS. Em teoria, este algoritmo é o mais eficiente comparativamente aos anteriores, pois contêm as vantagens da pesquisa em largura e da pesquisa em profundidade.

No caso do SameGame, se não se definir um limite mínimo inicial para este método ele começa com profundidade 0, o que significa que irá realizar a pesquisa em largura, ocupando demasiado espaço em memória para a maioria dos casos. Se for definido um limite mínimo mais razoável a pesquisa irá aproximar-se cada vez mais da rapidez do DFS.

O factor de complexidade temporal deste algoritmo é o mesmo que os anteriores, mas a sua complexidade espacial é de O(bd). Este método é o mais indicado quando não se sabe nem a dimensão da profundidade nem o grau de ramificação onde se poderá encontrar a solução.

A estrutura de dados deste método é a mesma que a do DFS.

Esquematização do algoritmo:

1. Define profundidade mínima = MIN_DEPTH
2. Chama DLS com profundidade limite = MIN_DEPTH
   - Se DLS encontrar solução termina
   - Senão incrementa MIN_DEPTH e repete 2.

5.2 Heurísticas utilizadas

Para melhor simular a inteligência artificial foi necessário encontrar fórmulas que, quando aliadas aos métodos de pesquisa, encaminhassem a pesquisa de solução para nós mais promissores, tentando maximizar o número de pontos obtidos no jogo.

Foi dispendido bastante tempo a pesquisar metodologias já existentes para este jogo, mas esta pesquisa revelou-se infrutífera visto que nenhum dos exemplos encontrados tinham a opção de ver o computador a jogar. Decidiu-se portanto criar as heurísticas de raiz, e embora inicialmente esta tarefa parecesse complicada, com o tempo veio a revelar-se a parte mais estimulante de todo o projecto.

Cada heurística, à excepção da primeira, usa as heurísticas anteriores para o cálculo do seu valor. Deste modo é assegurada uma ordenação mais racional das jogadas a pesquisar.

A qualidade atingida com a última heurística é bastante satisfatório, simulando diversas estratégias utilizadas por jogadores humanos e que de forma geral supera a prestação de um jogador humano. Existem no entanto casos pontuais em que as heurísticas mais complexas obtêm piores resultados que as heurísticas mais simples, mas isto deve-sse única e exclusivamente à aleatoriedade inerente ao SameGame.

Heurística 1 – Modo Fácil

A ideia base desta fórmula é calcular os pontos de cada grupo de bolhas a rebentar, organizando-as depois decrescentemente quanto ao valor obtido.

H1 = n*(n-1) Sendo n o número de bolhas a rebentar, por mancha de bolhas.

Heurística 2 – Modo Médio

Após se calcular a pontuação obtida por cada mancha é retirado o valor do número de bolhas da mesma que ainda se encontram isoladas no tabuleiro. Esta heurística visa prevenir jogadas em manchas que mais tarde no decorrer do jogo se poderiam revelar úteis, pois há sempre uma boa probabilidade de absorverem bolhas isoladas.

H2 = H1 – looseBubbles(cor) Sendo cor a cor da mancha em questão.

Heurística 3 – Modo Difícil

Esta heurística é a que usa os cálculos mais complexos, pois visita sequencialmente as bolhas vizinhas à mancha, na vertical, e no caso de serem da mesma cor calcula o valor da mancha que se irá formar ao ser destruída a mancha corrente. No final este valor será o somatório do valor de cada nova mancha criada desta maneira. Para além disto, são também somados os valores das coordenadas x e y da bolha mais próxima do canto inferior direito da mancha corrente, sendo que remover manchas no fundo e à direita do tabuleiro gera uma maior probabilidade de criar novas manchas.

$H3 = H2 + \sum n_i * (n_i - 1)$, Sendo n_i cada o número de bolhas de cada mancha resultante da eliminação da mancha corrente

Descrição do Algoritmo usado:

1. Para todas as bolhas da mancha
2. Percorrer verticalmente para cima até encontrar bolha de outra cor
3. Percorrer verticalmente para baixo até encontrar bolha de outra cor
    - Contar o tamanho das duas manchas em que se inserem estas bolhas (caso haja manchas)
    - Se são da mesma cor, atribuir pontuação
        - Se mancha anterior é da mesma cor, calcular pontuação da soma do tamanho das manchas
            - Iterar a mancha horizontalmente até encontrar um par de bolhas de cor diferente
        - Senão incrementa uma unidade horizontalmente e repete 1
    - Senão incrementa uma unidade horizontalmente e repete 1

Heurística 4 – Modo Nightmare!

Nesta última heurística, que se baseia em todas as anteriores, é adicionado um cálculo extra. Consiste basicamente em calcular a cor de maior frequência no tabuleiro, e atribuir posteriormente uma pontuação muito baixa às manchas desta cor que irão sendo

criadas ao longo da pesquisa. O objectivo é promover a formação de manchas muito grandes com esta cor. Jogadas que incidam em manchas da cor de maior ocorrência serão apenas efectuadas quando não existirem mais jogadas possíveis. Curiosamente, esta estratégia, apesar de simples, revelou ser a que melhores resultados traz no cálculo de soluções de puzzles Samegame, uma vez que a função de cálculo da pontuação oficialmente utilizada é polinomial de segundo grau. Isto significa que a pontuação cresce exponencialmente em função do tamanho da mancha.

H4 = H3 – F, sendo F o valor de desvalorização das manchas com a cor de maior ocorrência

Numa forma mais elegante:

$H4 = n*(n-1) – looseBubbles(cor) + \sum n_i * (n_i -1) + x + y – F(H4)$

6. Ambiente de Desenvolvimento

O jogo foi implementado usando a linguagem Java recorrendo à plataforma de desenvolvimento NetBeans IDE 6.0.1.

O desenvolvimento foi implementado num computador portátil com processador 1.6Ghz. 512MB de ram, sobre o sistema operativo Windows XP.

Foi utilizada a API Swing do Java, para a implementação da interface gráfica.

7. **Avaliação do programa**

As características a analisar para comparar o desempenho entre diferentes implementações do Samegame seriam:

- Robustez de implementação
- Funcionalidades
- Vários níveis de inteligência artificial
- Qualidade das soluções calculadas pelo computador
- Rapidez de cálculo de soluções

8. Resultados Experimentais

De forma a comparar e testar a validade das heurísticas foram efectuados inúmeros jogos de teste. A figura abaixo apresenta um exemplo destes testes efectuados, para um tabuleiro de 10x10 com 3 cores, usando o modo de pesquisa DFS (que com heurística é se torna Best-First Search).

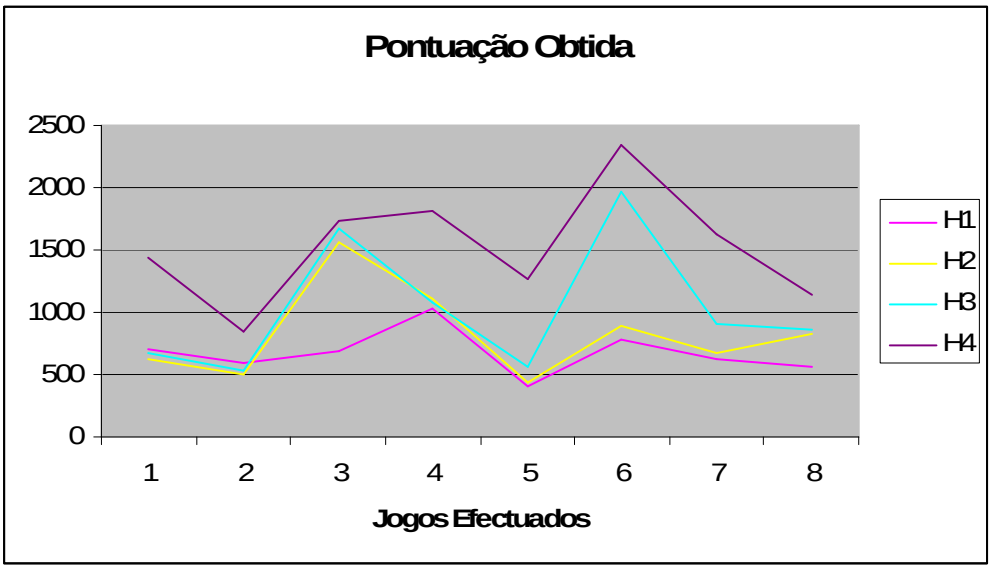

Figura 2 –Gráfico de comparação de resultados entre as quatro heurísticas

Como se pode verificar pelos dados, a última heurística é claramente a mais eficiente, obtendo nalguns casos resultados três vezes superiores ao das heurísticas 1 e 2.

Em relação ao número de jogadas os testes efectuados revelaram que a para uma pesquisa no modo DFS este valor é superior ao do modo BFS, para tabuleiros de pequenas dimensões. Iste deve-se ao facto de que a pesquisa em largura é a mais eficaz no que toca ao número mínimo de jogadas, que corresponde à profundidade mínima da árvore a gerar. Comparando as heurísticas entre si as conclusões que se podem tirar são semelhantes às das pontuações, ou seja, com uma heurística mais complexa o número de jogadas até encontrar a solução é mais baixo. Principalmente no caso da quarta heurística, pois ao se eliminar de uma só vez um grande número de bolhas, além de se pontuar bastante também se evita inúmeras jogadas.

Quanto aos tempos de execução dos algoritmos é possível distinguir claramente que a pesquisa em largura, BFS, nos casos em que encontra solução, é a mais lenta. A IDDFS varia conforme o valor definido para o limite mínimo, sendo que não definindo nenhum demora o mesmo tempo que a BFS. Deste modo, a pesquisa mais indicada para este problema é a DFS, visto que encontra uma solução quase que instantaneamente. Com as heurísticas ligadas as relações temporais entre as pesquisas mantêm-se. No entanto, se se seleccionar a opção de forçar uma solução em que o tabuleiro esteja vazio a IDDFS é a mais rápida para tabuleiros de pequenas dimensões.

Foram também testadas pesquisas DFS com heurísticas em tabuleiros com dimensões exageradas, tais como 50x50, sendo que o cálculo da solução pode demorar minutos para as heurísticas mais complexas. O facto de que uma solução é encontrada e de que a diferença na pontuação entre as heurísticas simples e complexas (por vezes na ordem das dezenas de milhar) apenas vem comprovar a robustez dos algoritmos implementados.

9. Conclusão

Após realizado este trabalho, provou-se que os métodos de pesquisa informada superam em muito as pesquisas tradicionais, tipicamente por força bruta. No caso do Samegame provou-se também que o melhor algoritmo de pesquisa consiste na pesquisa primeiro em profundidade, pois o factor de ramificação das árvores geradas é brutalmente exponencial. Foi especialmente interessante verificar a eficácia dos algoritmos implementados, que superam de uma forma geral a prestação de um jogador humano.

Possíveis melhoramentos a efectuar poderiam passar pelo estudo de outras heurísticas, à medida que se fossem descobrindo novas boas estratégias para a obtenção de bons resultados na resolução de puzzles.

Outro melhoramento passaria pelo embelezamento da interface da aplicação.

Apêndices

1. Exemplo de uma execução

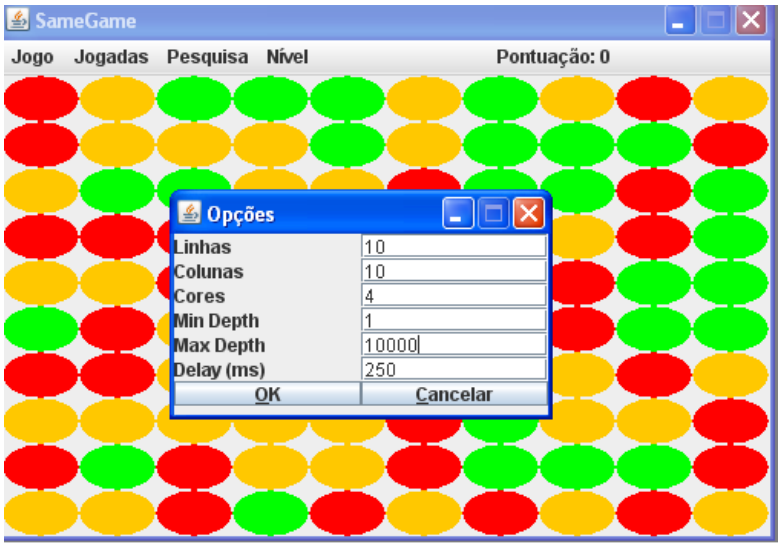

Figura 3 - Configuração

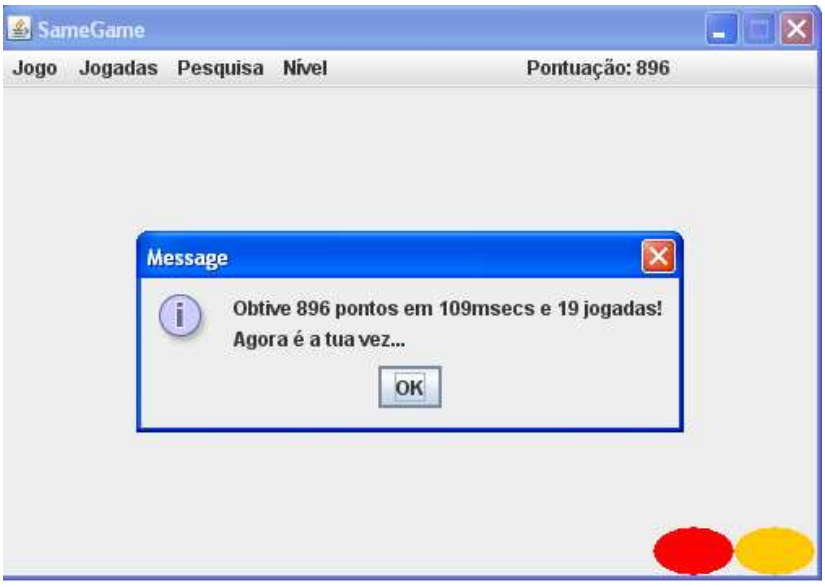

Figura 4 - Desafiar o computador

Figura 5 - Fim de jogo

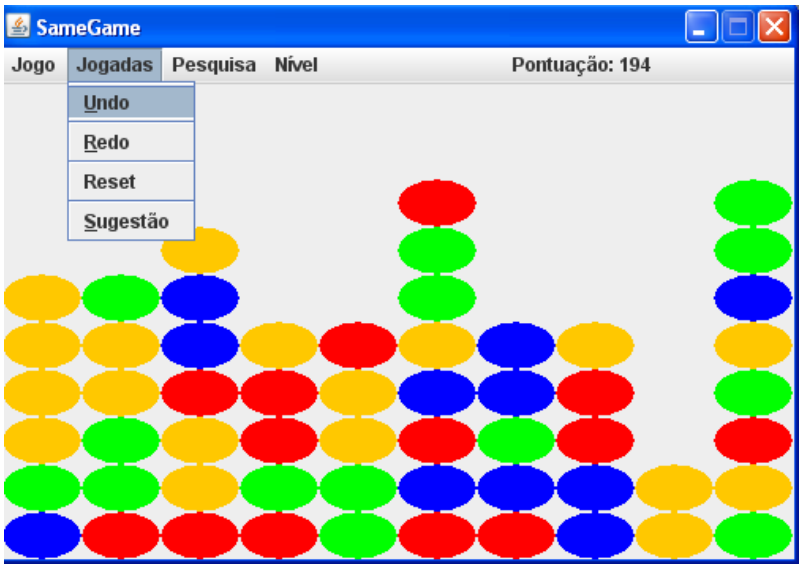

Figura 6 - Funcionalidades

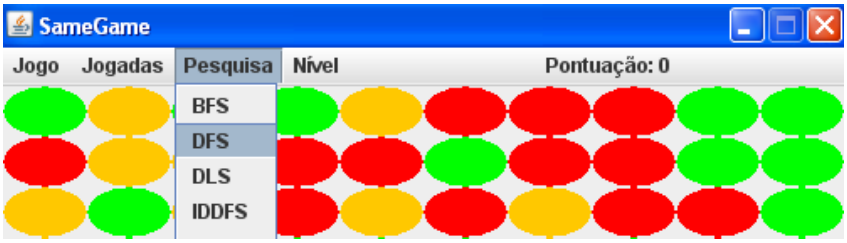

Figura 7 - Métodos de Pesquisa

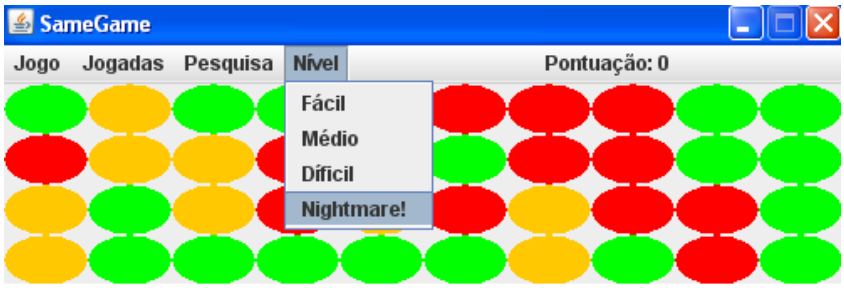

Figura 8 - Nível de dificuldade

Figura 9 - Sugestão de jogada

2. Referências Bibliográficas

Vários Autores, "SameGame", Wikipedia Foundation, Inc., 2008, disponível em: http://en.wikipedia.org/wiki/Samegame

Kari, J. "Samegame", ooPixel, 2008, http://www.oopixel.com/samegame/

Sun Developer Network, "JRE 1.6 API Specifications", Sun Microsystems, Inc., 2008, disponível em: http://developers.sun.com/

Sun Developer Network, "The Java Tutorial: Creating a GUI with JFC/Swing", Sun Microsystems, Inc., 2008, disponível em:
http://java.sun.com/docs/books/tutorial/uiswing/

Thornton, C., "AI Lecture 3 – Problem Solving", University of Sussex, UK, 2006, disponível em: http://www.cogs.susx.ac.uk/users/christ/crs/kr/lec03.html

Henkan, C., "Programming Languages – Principles and Paradigms: Notes on Iterative Deepening", Department of Computer Science and Engineering – University of California, San Diego, USA, 1999, disponível em:
http://www-cse.ucsd.edu/~elkan/130fall99/itdeep.html

"Introduction to AI Programming - Different Search Methods", Computing Department Lancaster University, UK, 1996, disponível em: http://www.comp.lancs.ac.uk/computing/research/aai-aied/people/paulb/old243prolog/subsection3_6_4.html

Vários Autores, "Breadth-first search", Wikipedia Foundation, Inc., 2008, disponível em: http://en.wikipedia.org/wiki/Breadth-first_search

Vários Autores, "Best-first search", Wikipedia Foundation, Inc., 2008, disponível em: http://en.wikipedia.org/wiki/Best-first_search

Vários Autores, "Depth-first search", Wikipedia Foundation, Inc., 2008, disponível em: http://en.wikipedia.org/wiki/Depth-limited_search

Vários Autores, “Depth-limited search”, Wikipedia Foundation, Inc., 2008, disponível em: http://en.wikipedia.org/wiki/Depth-limited_search

Vários Autores, “Iterative deepening depth-first search”, Wikipedia Foundation, Inc., 2008, disponível em:
http://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search

Vários Autores, “Heuristic”, Wikipedia Foundation, Inc., 2008, disponível em: http://en.wikipedia.org/wiki/Heuristic

Nielsen, J., “How to Conduct a Heuristic Evaluation”, Nielsen Norman Group, California, USA, 2008, disponível em:
http://www.useit.com/papers/heuristic/heuristic_evaluation.html

Lang, H.W., Flensburg, FH, “Sorting algorithms: Quicksort”, Impressum, Germany, 2008, disponível em:
http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/quick/quicken.htm