UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

ESTUDO COMPARATIVO ENTRE ALGORITMO A* E BUSCA EM LARGURA PARA PLANEJAMENTO DE CAMINHO DE PERSONAGENS EM JOGOS DO TIPO PACMAN

JEANITA BASSANI DA SILVA

BLUMENAU
2005

2005/1-27

JEANITA BASSANI DA SILVA

ESTUDO COMPARATIVO ENTRE ALGORITMO A* E BUSCA EM LARGURA PARA PLANEJAMENTO DE CAMINHO DE PERSONAGENS EM JOGOS DO TIPO PACMAN

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. Paulo César Rodacki Gomes, Doutor - Orientador

BLUMENAU
2005

2005/1-27

ESTUDO COMPARATIVO ENTRE ALGORITMO A* E BUSCA EM LARGURA PARA PLANEJAMENTO DE CAMINHO DE PERSONAGENS EM JOGOS DO TIPO PACMAN

Por

JEANITA BASSANI DA SILVA

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Paulo César Rodacki Gomes, Doutor – Orientador, FURB
Membro:	Prof. Maurício Capobianco Lopes - FURB
Membro:	Prof. Roberto Heinzle - FURB

Blumenau, 04 de julho de 2005.

Dedico este trabalho a meus pais pelo imenso amor e apoio a que sempre me dedicaram e por terem acreditado no meu sucesso.

AGRADECIMENTOS

À Deus, por ter me dado condições para a realização deste trabalho.

A meus pais, João e Jurema, pelo apoio, dedicação, amizade e incentivo que me deram para chegar até aqui.

À meu marido, Alessandro, por ter acompanhado todo o trabalho com paciência e amor.

Ao meu orientador, Paulo, pelo apoio, confiança, dedicação e amizade.

A meus amigos, agradeço por toda a amizade e companheirismo durante a vida acadêmica, que com certeza deixará saudades.

Vim aqui para ser todos e com todos, e o que faço hoje na minha solidão ecoará amanhã entre todos os homens.

Kahlil Gibran

RESUMO

Esta obra apresenta um estudo sobre Algoritmos de Grafos, abordando especificamente o seu uso para a representação e resolução de problemas de busca de menor caminho em jogos, relatando um estudo comparativo entre os algoritmos A*, busca em largura e busca em profundidade, demonstrando as suas particularidades e eficiência. Para validar o estudo, é apresentada a implementação de um jogo tipo PacMan com as buscas mencionadas, fazendo um comparativo com o desempenho de cada uma das buscas, com o objetivo de resolver o problema do menor caminho entre os personagens do jogo.

Palavras-chave: Jogos. Algoritmo A*. Busca em largura. Busca em profundidade.

ABSTRACT

This work specifically presents a study on Algorithms of Graphs, approaching its use for the representation and resolution of problems of search of lesser way in games, telling a comparative study it enters the algorithms A*, width search and deep search, demonstrating its particularitities and efficiency. To validate the study, the implementation of a game is presented PacMan type with the mentioned searches, making a comparative degree with the performance of each one of the searches, with the objective to decide the problem of the lesser way enters the personages of the game.

Word-key: Games. Algorithm A*. Width search. Deep search.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

ABNT – Associação Brasileira de Normas Técnicas

UML – *Unified Modelling Language*

SBC – Sociedade Brasileira de Computação

LISTA DE SÍMBOLOS

@ - arroba

% - por cento

$ - cifrão

(- abre parenteses

) – fecha parenteses

[- abre colchetes

] – fecha colchetes

“ – aspas dupla de abertura

” – aspas dupla de fechamento

* - asteristico

< - maior

> - menor

+ - soma

- - subtração

| - intersecção

/ - barra

$\in$ – pertence

= - igual

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos, cada vez mais os jogos por computador vêm implementando algum grau de inteligência em seus personagens. Um exemplo disto é a implementação de métodos de planejamento de caminho.

Este trabalho visa comparar técnicas utilizadas normalmente para planejamento de caminho de personagens de jogos de labirinto 2D, do tipo PacMan (o jogo pode ser encontrado em Postma (2000)). Estas técnicas são os algoritmos de busca heurística A*, a busca em largura e a busca em profundidade em grafos.

O jogo PacMan possui dois personagens, o “come-come” e os “fantasmas”. O jogador controla o personagem “come-come”, que tem o objetivo de comer todas as bolinhas do cenário do jogo. A fig. 1 a) mostra como é o desenho do personagem “come-come”. Os personagens “fantasmas” têm o objetivo de impedir a atividade do personagem “come-come” tentando encurralá-lo. A morte deste acontece quando algum “fantasma” consegue encostar-se a ele, impedindo que o jogador vença o jogo. A fig. 1 b) mostra como é o desenho do personagem “fantasma”. Uma ilustração de um cenário para o jogo pode ser visto na fig. 2.

	
a) personagem “come-come”	b) personagem “fantasma”

Figura 1 – Personagens

Em cada estado do jogo, o objetivo de cada “fantasma” é dirigir-se para o local onde está o “come-come”. Para isso, cada “fantasma” precisa calcular o caminho mais curto de sua posição até a posição do “come-come”.

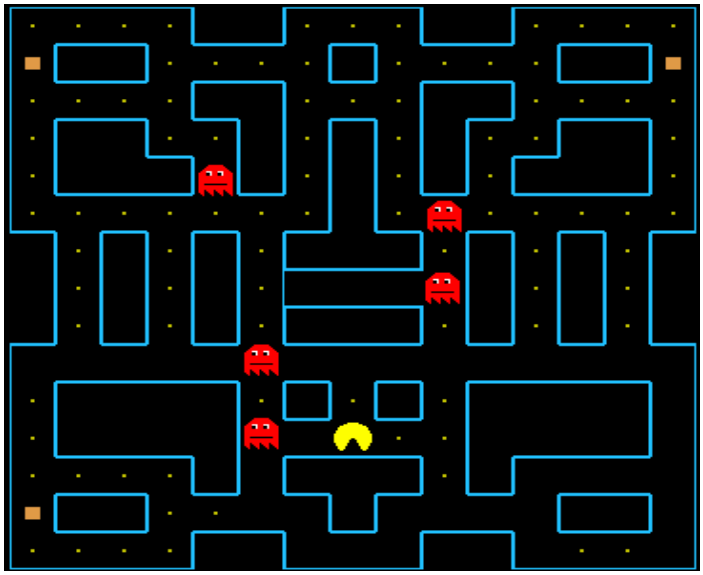


Figura 2 – Demonstração do jogo

A área onde se encontra o cenário do jogo pode ser modelada como um grafo, sendo que o cenário do jogo será uma adaptação do tradicional jogo PacMan, e o planejamento do caminho de cada “fantasma” resume-se a um problema de busca por caminho mínimo no grafo. Neste contexto, o caminho mínimo entre um par de vértices *u* e *v* em um grafo é definido como sendo o caminho com menor quantidade de arestas, dentre todos os caminhos possíveis entre esses dois vértices.

Devido à característica dinâmica dos jogos do tipo PacMan, acredita-se que os métodos tradicionais de busca possuem um desempenho muito lento, pois durante o processo de busca, eles visitam uma grande quantidade de vértices desnecessariamente.

Uma das técnicas que pode melhorar o desempenho dos personagens “fantasmas” em concluir o seu objetivo é o uso de um algoritmo de busca heurística. A heurística é uma técnica que melhora a eficiência de um processo de busca. No problema proposto, é necessária uma heurística que se aproxime bastante do caminho mínimo, mas que explore uma pequena quantidade de vértices para chegar à solução.

O algoritmo de busca heurística A* (RABUSKE, 1995, p. 41), a busca em largura (CORMEN et al., 2002, p. 422), e a busca em profundidade (CORMEN et al.,

2002, p. 429) encontram o menor caminho entre dois vértices em um grafo, podendo ser utilizados para planejamento do menor caminho entre o personagem “fantasma” e o personagem “come-come”.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é implementar um algoritmo de busca heurística, o A*, para encontrar o melhor caminho entre os personagens de um jogo, o PacMan, e comparar seu desempenho com o algoritmo de busca em largura e o algoritmo de busca em profundidade.

1.2 ESTRUTURA DO TRABALHO

No capítulo 2, o trabalho começa abordando conceitos sobre jogos de computador e sua história. Em seguida, é explicado como é resolvido o problema do caminho mínimo, utilizando-se de algoritmos de grafos e conceitos relacionados a eles. São analisados os principais algoritmos de grafos, como o busca em profundidade e o busca em largura. Realizando também um estudo sobre a busca heurística, como sendo uma técnica que utiliza do conhecimento para a resolução de problemas. Serão apresentados os algoritmos A*, o A* adaptado e o busca em profundidade adaptado. Estes dois últimos são adaptações do algoritmo A* específicas para o problema proposto.

No capítulo 3, é explicado como foi realizado o desenvolvimento do protótipo de jogo, onde são detalhados os métodos utilizados, apresentando os principais requisitos do problema e a especificação através dos diagramas de casos de uso,

diagrama de classes e diagrama de seqüência, demonstrando como foi realizada a implementação através de suas técnicas e ferramentas e também a operacionalidade de implementação, finalizando com um comparativo dos resultados obtidos.

Por fim, o capítulo 4 apresenta as considerações finais e sugestões para trabalhos futuros.

2 ESTUDO COMPARATIVO ENTRE ALGORITMOS DE CAMINHAMENTO EM GRAFOS

Basicamente, os algoritmos de caminhamento em grafos são utilizados para a resolução de vários problemas de caminho mínimo. No contexto desta obra é feito um estudo sobre sua aplicação relacionada a jogos, mais precisamente a jogos tipo PacMan. Para uma melhor compreensão do trabalho apresentado, neste capítulo serão apresentados conceitos e estruturas básicas de relevância para o tema proposto.

2.1 JOGOS POR COMPUTADOR

Segundo Battaiola (2000, p. 83), os jogos por computador têm apresentado um surpreendente grau de evolução tecnológica nos últimos anos. O grande interesse no desenvolvimento desse tipo de software se deve ao seu atrativo comercial, cujo mercado mundial se situa na faixa das dezenas de bilhões de dólares.

2.1.1 Evolução histórica dos jogos por computador

Para se entender as características atuais dos jogos por computador, como também a influência que eles tiveram no desenvolvimento de diversas tecnologias, é importante conhecer a sua evolução histórica (Battaiola, 2000, p. 84).

A descrição a seguir enfoca os jogos por console, aqueles em que o usuário compra um console que vem com um conector para o aparelho de TV e um encaixe para algum tipo de unidade de armazenamento, que contém o programa e/ou os dados de um jogo particular.

De acordo com Battaiola (2000, p. 87), em termos de jogos por console, a

evolução pode ser sintetizada pelo histórico abaixo:

Em 1972 o *Odyssey*, da Magnavox, foi colocado no mercado americano ao preço de US$ 100,00. Sendo que circuitos integrados eram caros na época, este console foi construído usando somente de poucos e simples efeitos na tela. Para tornar o jogo mais interessante, a unidade era empacotada com 2 controles e outros acessórios diversos (dados, dinheiro de brinquedo, roletas, cartas, etc.). Em especial, o jogo vinha com algumas coberturas plásticas para o monitor da TV, cuja aplicação era simular um ambiente de um jogo mais complexo. O jogador tinha que anotar o placar, porque a máquina não era capaz de fazer isso.

Em 1977 o *Vídeo Computer System* (VCS), da Atari, foi liberado ao preço de US$ 200,00 e, pelo seu grande sucesso, foi comercializado até 1990. A parte central do sistema era um processador Motorola 6507 de 8 bits e 1.19 Mhz e 256 bytes de RAM. Foi o primeiro console a permitir jogos coloridos e as imagens se limitavam a pontos e traços.

Em 1982 o *Colecovision*, da Coleco, continha 48 Kb de RAM de memória, um microprocessador Z-80A de 8 bits e 3.58 Mhz e custava US$ 200,00. O fabricante planejava vários módulos de expansão para o sistema, entre eles, um em que o transformava em um computador pessoal.

Em 1989 o *Game Boy*, da Nintendo, um sistema manual portátil, usava uma tela de cristal líquido preta e verde e era programável a partir de cartuchos passíveis de troca. Ele continha um microprocessador de 8 bits e 1.1 Mhz, custava US$ 100,00 e foi um grande sucesso.

O Lynx, da Atari, foi criado pelos projetistas do computador pessoal Commodore Amiga, que decidiram criar o primeiro jogo portátil manual colorido. O jogo operava com um microprocessador de 8 bits e uma tela suficientemente grande e

capaz de exibir imagens coloridas com detalhes finos. Foi comercializado a partir de US$ 149,00.

Em 1995 foi lançado o *Playstation*, da Sony. Este equipamento originou-se de uma conexão de CD ao Super NES. Desacordos entre a Sony e a Nintendo sobre a forma de comercialização do sistema, fizeram com que a Sony decidisse por desenvolver a sua própria máquina, no caso, o Playstation, que contava com microprocessador de 32 bits e 33 Mhz, especialmente projetado para produzir gráficos poligonais.

Em 1996 o *Nintendo* 64 incorporou a tecnologia de supercomputadores da década de 80 através do uso de processadores de 64 bits e 93.75 Mhz. O projeto foi feito em conjunto pela Nintendo e a Silicon Graphics. O sistema de controle, por si só, já era revolucionário, pois foi projetado especificamente para jogos 3D. O console foi comercializado por US$ 150,00.

Em março de 2000, é lançado o Playstation 2, da Sony, no Japão ao preço de aproximadamente US$ 370,00. Ele é equipado com um processador de 128 bits e 294.91 Mhz, com memória de 32 Mb, três co-processadores, um de ponto flutuante e dois vetoriais, atingem a performance de 6.2 Gflops em cálculos vetoriais. Com iluminação processa 34 milhões de polígonos por segundo. Tem encaixe para CD e DVD e pode exibir filmes em DVD.

2.1.2 Definição de um jogo por computador

Segundo Santos (2002, p. 8), um jogo é um programa de computador que se caracteriza por intensa interatividade e participação do usuário ou jogador, utilizando sua habilidade com os dispositivos de entrada, seu raciocínio lógico e sua imaginação

para alcançar um determinado objetivo. O objetivo do jogo pode ser uma tarefa simples, como chegar em primeiro lugar em uma corrida ou eliminar do tabuleiro todas as peças do adversário.

Os jogos, em geral, utilizam recursos dos dispositivos de saída, como desenhos no monitor e ruídos através da placa de som, para mostrar ao jogador o estado corrente do jogo, para orientá-lo e estimulá-lo à conquista do objetivo. Jogos bem elaborados possuem história intrincada, personagens marcantes e objetivos misteriosos.

> Os jogos necessitam de estruturas de dados para representar todos os seus elementos: um jogo de corrida de carros precisa armazenar a posição ocupada por cada carro em cada momento da corrida; um jogo de damas deve registrar as posições de todas as peças do tabuleiro; um jogo de ação ou aventura precisa guardar a posição do jogador, dos inimigos e das demais entidades. De acordo com o grau de complexidade do jogo, pode ser necessário armazenar informações sobre a posição de diversos objetos dentro do espaço do jogo. (SANTOS, 2002, p. 8).

Segundo Rollings (2003, p. 8), um jogo para computador pode ser definido como um sistema composto por três partes básicas: enredo, motor e interface interativa. O sucesso de um jogo está associado à combinação perfeita destes componentes. O enredo define o tema, a trama, os objetivos do jogo, o qual através de uma série de passos o usuário deve se esforçar para atingir. O motor é o seu sistema de controle, o mecanismo que controla a reação do jogo em função de uma ação do usuário. A interface interativa controla a comunicação entre o motor e o usuário, reportando graficamente um novo estado do jogo.

2.1.3 Roteiro de um jogo

Segundo Battaiola (2000, p. 93), a estrutura básica do roteiro de um jogo é composta de quatro itens: sinopse, personagens, cenários e comentários finais. Cada

um deles é abordado a seguir.

Na sinopse é sintetizada toda a trama do jogo. No caso de um jogo baseado em um enredo cinematográfico, a sinopse também deve conter uma análise da ação que transcorre do primeiro ao terceiro ato, no mínimo. Esta parte do jogo deve ser elaborada prioritariamente pelo idealizador do jogo.

O item personagem consiste na descrição das características centrais de cada personagem, divididas em: descrição sucinta de suas características e papel na trama (“nobre, atlético e inteligente que salvará a princesa em seu castelo”), nome, idade, aparência (“jovem musculoso que se veste com roupas de grife”), equipamentos que opera (“revólver, canhão laser, etc.”), caráter (“gentil, defensor dos fracos e oprimidos”) e origem.

No cenário é descrito, para cada etapa do jogo, o local físico onde ocorre a ação, a era, o clima, a temperatura, etc. Se um determinado local é habitado por fantasmas ou monstros, ou então escondem armas ou armadilhas, todos estes elementos devem constar na descrição daquele cenário em particular.

Os comentários finais contêm observações e sugestões variadas. Em especial, pode haver sugestões de como o final do enredo pode ser a ponte para um novo jogo, de como transformar o personagem principal em um ator de desenho animado ou história em quadrinho, de como associar uma música de sucesso ao jogo para ressaltar sua trilha sonora, etc.

2.1.4 Tipos de jogos

Existem diversos tipos de jogos, dentre eles estão os jogos *arcades* que possuem interface simples, os jogos de aventura que eram baseados em texto, os jogos

de aventura gráficos que são os sucessores dos jogos de aventura baseados em texto e os jogos de estratégia e de guerra, que foram inspirados em jogos de tabuleiro. A seguir eles são apresentados em maiores detalhes.

2.1.4.1 Jogos *arcades*

Segundo Rollings et al (2003, p. 151), inicialmente as interfaces e os jogos eram simples. Para jogos estilo *arcade*, deveria haver uma tela de título com instruções ou duas a três telas que se repetiam uma após outra (título, instruções e tabela de *scores*). O jogo ficaria nesse estado até que um jogador pressionasse o botão de início. Uma vez iniciado o jogo, a interface era usualmente muito simples. Haveria vários *displays* como: pontuação atual, nível do jogo (se preciso) e vidas restantes. Esses *displays* seriam colocados no topo ou no rodapé da tela. Por muitos anos, esta foi a interface dos jogos a*rcades* (mostrado na figura 3). Naturalmente, houve muitas variações, por exemplo, um medidor de nível de energia em vez do *display* de vidas restantes.

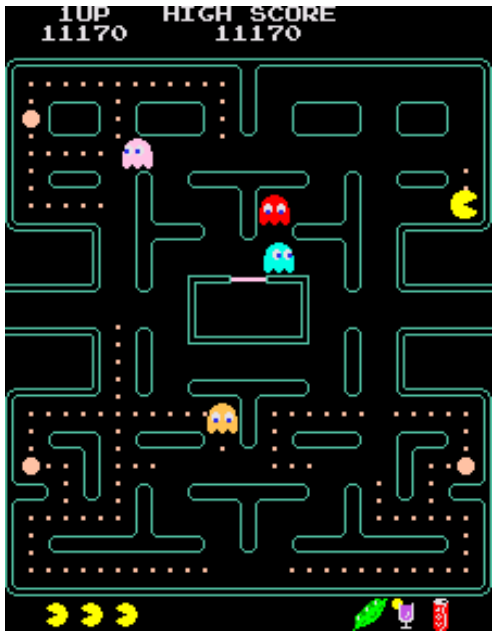


Figura 3 – O jogo padrão de interface a*rcade* – PacMan Plus

2.1.4.2 Jogos de aventura

Segundo Rollings et al (2003, p. 155), tradicionalmente, os jogos de aventura (mostrado na figura 4), eram baseados em textos. O jogador interagia com o sistema lendo uma descrição textual da situação, e montava ações baseadas naquela descrição. O aventureiro colocava a informação em formato “verbal”. Por exemplo: “pegue-comida” funcionaria, entretanto “pegue comida em cima da mesa” seria rejeitado. Comando de movimento geralmente eram palavras simples – cima, baixo, norte, sul, leste e oeste – e poderiam ser abreviados para c, b, n, s, l, o, etc.

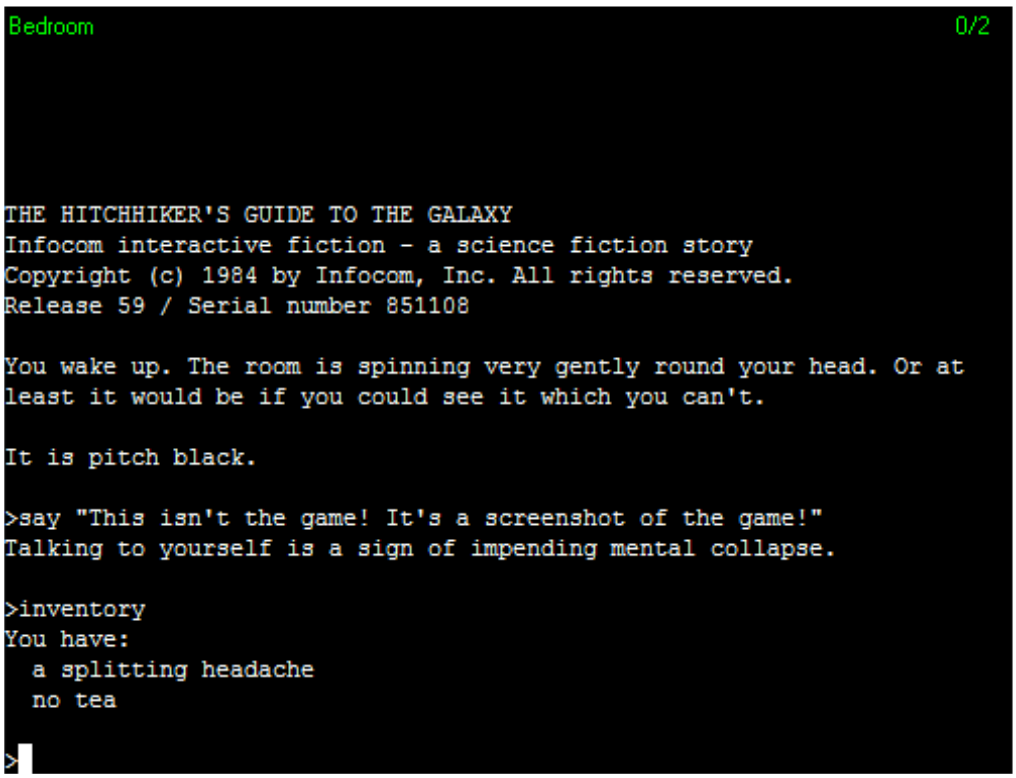


Figura 4 – *The Hichhiker´s Guide to the Galaxy*

2.1.4.3 Jogos de aventura gráficos

Segundo Rollings et al (2003, p. 156), os jogos de aventura gráficos são os sucessores dos jogos de aventura baseados em texto. Com a máxima “uma figura vale

mais que mil palavras” como discurso, os *designers* começaram a ter vantagem com o crescimento do poder dos computadores para criar uma interface inteiramente gráfica para os jogos de aventura. A interface apontar e clicar dos jogos de aventura gráficos teve pequenas mudanças desde que surgiu. Desde um dos primeiros jogos (mostrado na figura 5 – *Leisure Suit Larry II*) aos jogos atuais (mostrado na figura 6 – *Escape from Monkey Island*), a interface permanece amigavelmente consistente e razoavelmente simples de se construir: muitos gráficos desses jogos são 2D ou pseudo-3D, ou seja, usam gráficos 3D em um cenário 2D, como um palco.

Figura 5 – *Leisure Suit Larry* II

Figura 6 – *Escape from Monkey Island*

2.1.4.4 Jogos de estratégia e de guerra

Segundo Rollings et al (2003, p. 161), esses jogos foram inspirados em jogos de tabuleiro. Todos os jogos deste tipo usam basicamente as mesmas ações: controlar grandes grupos de unidades para atingir uma meta que não poderia ser atingida por apenas uma unidade. Uma vez que se vai movimentando o personagem através da superfície do jogo, há uma variação na interface do usuário: controles e decisões específicas, por exemplo no jogo *Civilization* III (mostrado na figura 7) que além de decisões para unidades e grupos possui um controle de diplomacia.

Figura 7 – *Civilization* III

2.2 PROBLEMA DE CAMINHO MÍNIMO

Para resolver o problema de caminho mínimo entre personagens de um jogo tipo PacMan são utilizados algoritmos de grafos. Para uma melhor compreensão do

trabalho, são mostrados a seguir principais conceitos de algoritmos de caminhamento em grafos.

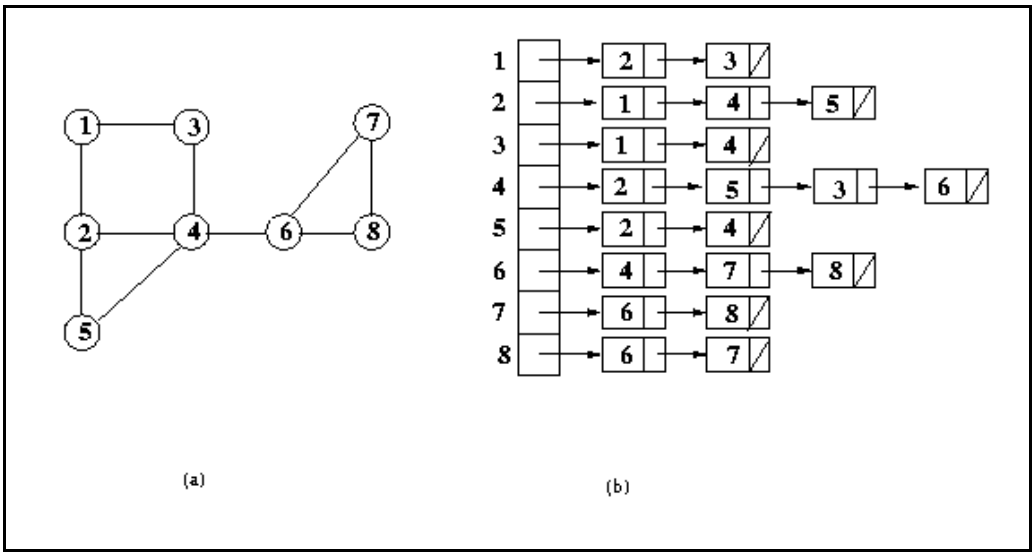
2.2.1 Algoritmos de grafos

Segundo (CORMEM, 2002, p. 419), os grafos são estruturas de dados sempre presentes na ciência da computação, e os algoritmos para trabalhar com eles são fundamentais na área. Existem centenas de problemas computacionais interessantes definidos em termos dos grafos.

Um grafo G = (V, E) é um conjunto de finito não vazio V e um conjunto E de pares não ordenados de elementos distintos de V. Os elementos de V são os vértices e os de E são as arestas de G respectivamente. Cada aresta *e* ϵ E é denotada pelo par de vértices e = (v,w) que a forma. Nesse caso, os vértices *v* e *w* são os extremos da aresta *e*, sendo denominados adjacentes. A aresta *e* é dita incidente a ambos *v* e *w* (CORMEN, 2002, p. 419).

2.2.2 Listas de adjacência

Um grafo G = (V, E) é representado por uma coleção de listas de adjacência ou como uma matriz de listas de adjacência. No presente trabalho será utilizada a representação de lista de adjacências, que consiste em um arranjo de adjacência de $|V|$ listas, uma para cada vértice em V. Para cada *u* ϵ V, a lista de adjacências Adj. [u] contém (ponteiros para) todos os vértices *v* tais que existe uma aresta (u, v) ϵ E (CORMEN, 2002, p. 420).



Fonte: adaptado de Cormen (2002, p. 420).

Figura 8 – a) grafo não orientado que tem oito vértices e nove arestas. b) Grafo não orientado como uma lista de adjacências.

2.3 BUSCA EM GRAFOS

Vários problemas representados por um grafo podem ser resolvidos efetuando uma busca nesse grafo. Às vezes é preciso visitar todos os vértices de um grafo, ou também o problema pode ser resolvido visitando-se somente um subconjunto de vértices. Considera-se, por exemplo, o problema do caminho mínimo. Os algoritmos apresentados para resolver esse problema fazem um percurso exaustivo de todos os vértices. Não precisa ser assim se, por exemplo, pretende-se determinar o caminho mais curto até um vértice em particular. Nesse caso, assim que ele se encontra no conjunto dos vértices já visitado, não é preciso continuar o algoritmo.

Dentre as técnicas existentes para a solução de problemas algorítmicos em grafos, a busca ocupa lugar de destaque, pelo grande número de problemas que podem ser resolvidos através da sua utilização. A busca consiste em localizar um vértice dentro de um grafo, traçando o caminho percorrido para se deslocar de um vértice inicial até o vértice que se deseja procurar.

Dentre os métodos existentes são apresentadas: a busca em profundidade e a busca em largura.

2.3.1 Busca em profundidade

A busca em profundidade é um método de busca que funciona utilizando uma pilha de vértices *(Last In First Out*, isto é, o último que entra é o primeiro que sai). Esta técnica explora o caminho para o objetivo, dando preferência aos vértices que estão mais distantes da raiz da árvore de busca.

Neste contexto, as árvores de busca são criadas baseadas nas alternativas de jogadas. A árvore de busca é um método usado para pesquisar um elemento em particular dentro de uma grande quantidade de dados de forma eficiente. A árvore de busca (exemplos de árvores de busca na figura 9) tem uma relação entre os nós ou como se referem nesta obra, vértices. Cada nó da árvore de busca pode ter no máximo um nó filho à esquerda e um nó filho à direita, sendo que cada um representa uma situação do jogo.

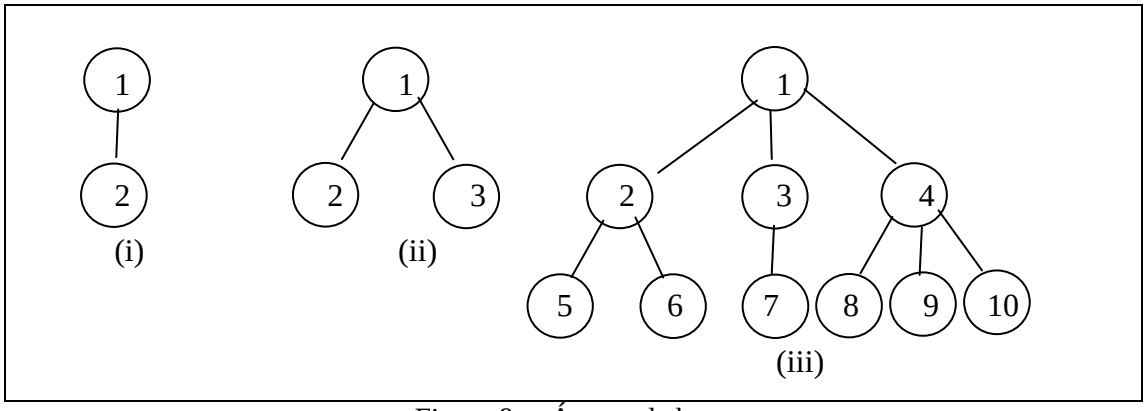


Figura 9 – Árvore de busca

Seja um grafo G = (V,E) que contém *n* vértices. Seja também uma representação que indica, para cada vértice, se ele foi visitado ou não. O algoritmo mostrado no Quadro 1 demonstra a busca em profundidade.

```
Procedimento Busca(g: Grafo)
     Para Cada vértice v de g:
         Marque v como não visitado
     Para Cada vértice v de g:
          Se v não foi visitado:
             Busca_Profundidade(v)

Procedimento Busca_Profundidade(v: vértice)
     Marque v como visitado
     Para Cada vértice w adjacente a v:
         Se w não foi visitado:
             Busca_Profundidade(w)
```

Quadro 1 – Algoritmo de busca em profundidade

Para implementar este algoritmo, é utilizada a estrutura de listas de adjacência. Isso porque a principal operação efetuada pelo algoritmo é a escolha de um vértice adjacente e sabe-se que nesse caso a estrutura de adjacência é a melhor opção. Supondo então que a estrutura de adjacência é utilizada para implementar a busca, pode-se ver que o tempo de execução do algoritmo é em O(max(a,n)), onde a e n representam o número de arestas e de vértices, respectivamente. Como cada vértice deve ser visitado, o algoritmo necessariamente fará n chamadas do procedimento Busca_Profundidade, muitas delas recursivas. Em cada chamada, todos os vértices adjacentes serão testados. No total, o número de testes realizados será igual ao número de arestas. Então o tempo total gasto para as chamadas a Busca_Profundidade é em O(a). A esse tempo deve-se acrescentar o tempo gasto no procedimento Busca: o tempo de inicialização, e o tempo para testar, para cada vértice, se ele foi marcado. No total isso dá um tempo em O(2n) = O(n). Portanto, a busca em profundidade tem um tempo de execução de ordem O(a+n). Na verdade têm-se um tempo em O(max(a,n)): se o grafo tem mais arestas que vértices têm-se um tempo em O(a). No caso contrário, têm-se um tempo em O(n).

Note que à busca em profundidade corresponde uma árvore geradora. No

exemplo da figura 8, obtêm-se a árvore gerada ilustrada na figura 10:

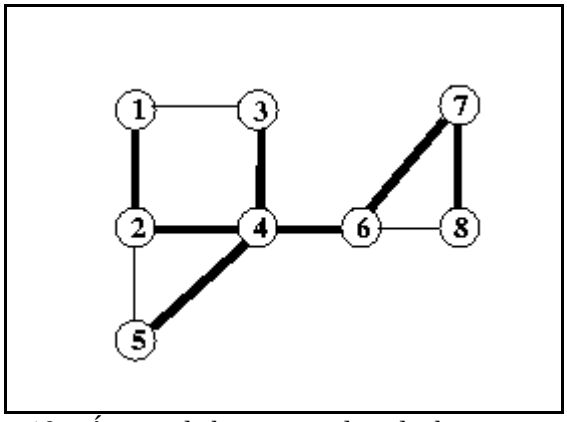


Figura 10 – Árvore de busca gerada pela busca em profundidade.

2.3.2 Busca em largura

A busca em largura é um método de busca que utiliza uma fila para determinar a ordem de visitação dos vértices *(First in Firt Out*, isto é, o primeiro que entra é o primeiro que sai). Esta busca examina todos os vértices de certo nível do grafo antes dos vértices do nível abaixo. Se o grafo é finito e se existe uma solução, então ela será encontrada por este método.

Em uma busca em largura a partir de um vértice *v*, espera-se que todos os vizinhos de *v* sejam visitados antes de continuar a busca mais profundamente. O algoritmo mostrado no Quadro 2 ilustra de forma suscinta este tipo de busca.

```
Procedimento Busca(G: Grafo)
         Para cada vértice v de G:
             Marcar v como não visitado;
         Para cada vértice v de G:
             Se v não foi visitado:
                 Busca_Largura;

Procedimento Busca_Largura(v: vértice)
         Inicializar Fila;
         Marcar v como visitado;
         Colocar v no final de Fila;
              Enquanto Fila não vazio:
                 u := primeiro elemento de Fila;
                 Retirar u de Fila;
                 Para cada vértice w adjacente a u:
                       Se w não foi visitado:
                           Marcar w como visitado ;
                           Colocar w no final de Fila;
```

Quadro 2 – Algoritmo de busca em largura.

Aplicando o algoritmo de busca em largura ao grafo da figura 8, pode-se obter a árvore geradora ilustrada na figura 11.

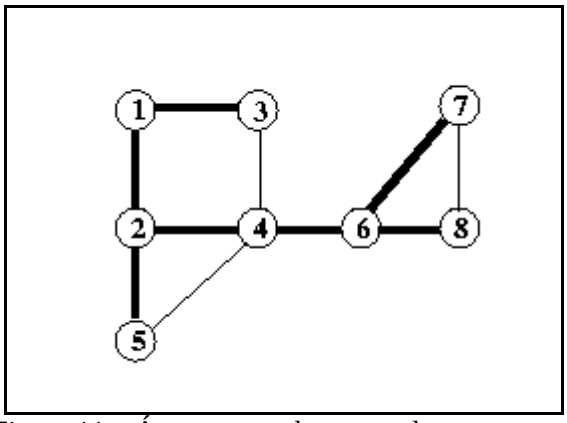


Figura 11 – Árvore que a busca em largura gerou.

O tempo de execução da busca em largura é o mesmo que o tempo de execução da busca em profundidade: O(n+a), ou O(max(n,a)). Mas, considerando o uso de memória, o desempenho é pior. Na busca em profundidade, somente um "ramo" da árvore é empilhada em qualquer momento. Então a pilha não conterá mais de n elementos.

2.4 BUSCA HEURÍSTICA

De acordo com Russel e Norvig (2003, p. 94), a busca heurística é uma estratégia que utiliza o conhecimento específico do problema, podendo encontrar soluções de forma mais eficiente que uma estratégia sem informação.

Segundo Rich e Knight (1993), a heurística é uma técnica eficiente em um processo de busca. A heurística é útil, pois leva a direcionamentos interessantes, mesmo que possa ignorar pontos de interesse para alguém. Em geral, as heurísticas melhoram os caminhos a serem percorridos, porém pode ocorrer de ignorar o melhor caminho. Pode-se obter boas soluções usando uma boa heurística em problemas difíceis.

Dentre os métodos existentes são apresentados: o algoritmo A*, o algoritmo A* adaptado e o busca em profundidade adaptado.

2.4.1 O algoritmo A*

Segundo Rabin (2002, p. 105), o algoritmo A* deve encontrar um caminho entre dois vértices em um grafo. Existem muitos algoritmos de busca, mas o A* encontra o menor caminho, se existir, e fará isto relativamente rápido – é isto que o diferencia dos outros.

O algoritmo A* atravessa o grafo marcando vértices que correspondem às várias posições exploradas. Esses vértices gravam o progresso da busca. Além de guardar a posição do vértice em um espaço 2D, cada um desses vértices têm 3 atributos principais chamados comumente de F, G e H:

a) O atributo G de um determinado vértice é o custo que se tem desde o vértice de inicio até este vértice destino;
b) O atributo H de um vértice é o custo estimado do caminho deste vértice até o vértice final. H entende-se por heurística e significa “estimativa bem comportada”, visto que realmente não se conhece esse custo;
c) O atributo F é a soma de G e H. F representa a melhor avaliação para o custo de um caminho passando através de um determinado vértice. Quanto menor o valor de F, para um determinado vértice, maior as chances de ele fazer parte do menor caminho.

O objetivo de F, G e H é quantificar o quão promissor é o caminho passando por determinado vértice. O atributo G pode ser calculado. Ele é o custo para chegar ao vértice corrente. Desde que se tenha explorado todos os vértices que apontam para este, sabe-se o valor exato de G. Entretanto, o atributo H é completamente diferente. Desde que não se sabe quanto além está o vértice final depois deste, deve-se estimar este valor. Na melhor das avaliações, o F fica mais próximo do valor correto, e mais rápido o A* encontra o fim com um pequeno esforço de empenho (RABIN, 2002, p. 106, tradução nossa).

O Quadro 3 mostra o algoritmo A* em pseudo-código.

```
// c(u,v) é o custo associado à aresta que liga u a v
// h(v) é a distância euclidiana do vértice v ao destino

A_Estrela(G,s,t)
 f[],g[],pai[]: vetor de inteiros
 Abertos, Fechados: conjunto de vértices
 achou: booleano
 novo_f: inteiro
 g[s] := 0
 f[s] := g[s] + h(s)
 pai[s] := nulo
 Abertos := vazio
 inclui s em Abertos
 Fechados := vazio
 achou := falso
 enquanto Abertos não é vazio e não achou faça
       v := melhor_vértice(Abertos)
       se v = t então
             achou := verdadeiro
       fim se
      para cada vértice u vizinho de v faça
            novo_f := g[v] + c(v,u) + h(u)
             se ((u está em Fechados) ou (u está em Abertos)) e
            (novo_f >= f[u]) então
                  continua /* processa o próximo vizinho */
            senão
                  pai[u] := v
                   g[u] := g[v] + c(u,v)
                  f[u] := novo_f
                   se u está em Fechados então
                         remove u de Fechados
                   fim se
                   se u está em Abertos então
                        remove u de Abertos
                   fim se
                   inclui u em Abertos
             fim se
       fim para
       inclui v em Fechados
 fim enquanto
      se achou então
       retorna sucesso
 retorna fracasso
```

Quadro 3 - Algoritmo A* utilizando distância euclidiana como heurística subestimada

Como é demonstrado no quadro 3, o algoritmo A* mantém duas listas de vértices, uma para vértices abertos e outra para fechados. O vértice raiz é o primeiro a entrar na lista de abertos, e seu custo é marcado como zero. Como ele é o melhor e o único vértice da lista ele é o primeiro a ser analisado.

Diante do problema proposto, o jogo PacMan possui um labirinto, que é o cenário onde o jogo acontece. O labirinto do jogo é modelado como um grafo representado por listas de adjacência, sendo que cada vértice do grafo possui uma coordenada x e y que é a posição do vértice, portanto a distância de custo mínimo é a função da distância geométrica entre os vértices.

A análise é feita da seguinte forma, pegam-se todos os vértices adjacentes e calcula-se o custo para chegar até o vértice de destino. Este custo é calculado da seguinte forma: distância cartesiana entre o vértice e o destino + custo adicional + distância calculada do vértice pai.

Foi utilizada a distância cartesiana entre o vértice e o destino ao invés da euclidiana como é utilizada normalmente pelos algoritmos A*, pelo fato do jogo em que é aplicado este algoritmo ser em um ambiente 2D, a distância euclidiana definiria o menor tamanho de uma linha reta da raiz até o destino, enquanto a cartesiana define o exato número de vértices entre um e outro e ainda exige menos processamento, visto que é feita usando apenas soma e subtração. Sendo assim a fórmula para o cálculo da distância cartesiana é: |DestinoX - OrigemX| + |DestinoY-OrigemY|, enquanto para o cálculo da distância euclidiana é: (RaizQuadrada((|DestinoX - OrigemX|)^2 + (|DestinoY-OrigemY|)^2)).

O vértice pai de um determinado vértice u é o vértice anterior a u no caminho que está sendo construído. A distância calculada do vértice pai, ou custo calculado, é o valor F=G+H do vértice pai. No caso do vértice raiz, este valor é zero, e no próximo será a distância já calculada. Essa distância é acrescentada para definir corretamente o custo total do caminho a ser percorrido, se o algoritmo tiver que escolher entre virar a esquerda com custo 30, e a direita com custo 20, ele irá virar para a direita.

Uma vez calculada a distância, o vértice é colocado na lista de abertos. Depois de ter todos os vértices vizinhos testados, o melhor vértice, que saiu da lista de abertos vai para a lista de vértices fechados, e não será testado novamente a não ser que ele seja vizinho de outro vértice que esteja sendo testado. Os vértices na lista de abertos já tiveram seus custos estimados, e dentre eles será escolhido sempre o de menor custo. Quando o melhor é escolhido, o procedimento acima é repetido, até que se encontre o vértice de destino ou até que não existam mais vértices na lista de abertos para testar.

2.4.2 O algoritmo A* adaptado

Este algoritmo, A* adaptado, é proposto pela autora do presente trabalho com o objetivo de otimizar a busca A* devido a particularidades do problema aqui abordado . A adaptação do A* traz uma única diferença funcional é que corresponde a limitar o número de vértices que podem ser tirados da lista de abertos. Ele testa até 25 vértices e quando chega nesse número, o algoritmo pára de procurar, independente de ter encontrado o vértice de destino ou não. Foi escolhido o número 25, pela quantidade de vértices adjacentes que o algoritmo A* original testa, e também devido ao tamanho do labirinto implementado para o jogo. Isso o aperfeiçoa em velocidade e em número de vértices a serem testados no total, porém nem sempre encontra o menor caminho.

2.4.3 Busca em profundidade adaptado

O algoritmo busca em profundidade adaptado, também proposto pela autora do presente trabalho, utiliza a mesma base do algoritmo de busca em profundidade já exemplificada anteriormente, com a diferença de que ao invés de testar os vértices na ordem

da lista de adjacências, ele faz o teste na ordem de menor custo, mesclando o busca em profundidade com o A*.

O menor custo é incrementado da seguinte forma: cada vez que um vértice é visitado pela busca e o vértice visitado não é a solução do problema, ele incrementa um custo adicional x, fazendo com que a busca escolha um caminho de menor custo, ou seja, caminho este que ainda não foi testado.

2.5 TRABALHOS CORRELATOS

Santos (2002) desenvolveu um trabalho sobre Inteligência Artificial aplicada a jogos em três dimensões. Em especial, é dada ênfase à camada de busca de caminhos, e é apresentada uma implementação do algoritmo A* para encontrar caminhos no espaço 3D. Foi implementado um algoritmo de suavização do caminho gerado através de curvas de Bézier, buscando diminuir as descontinuidades no movimento de qualquer entidade que se locomova ao longo desse caminho.

Lester (2003) em um artigo descreve os conceitos básicos do algoritmo A*, demonstrando a sua eficiência na busca de caminhos. São apresentados exemplos da implementação do algoritmo A* na linguagem C++.

Matthews (2000) realizou um estudo que demonstra a utilização do algoritmo A* buscando um caminho em um labirinto simples, discutindo também a teoria do A*.

Pereira, (1999) desenvolveu uma implementação para determinação do caminho de menor custo, baseado em um mapa, comparando o desempenho de algoritmos de busca cega com algoritmo de busca heurística.

3 DESENVOLVIMENTO DO PROTÓTIPO DE JOGO

Neste capítulo serão detalhados os métodos utilizados para o desenvolvimento do trabalho proposto, apresentando passo a passo para uma melhor compreensão, bem como os seus resultados, fazendo um comparativo do que foi estudado/desenvolvido.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Será apresentado um estudo de caso para melhor compreensão do trabalho proposto, bem como os requisitos funcionais e o requisito não funcional.

3.1.1 Estudo de caso

O jogo PacMan possui dois personagens o “come-come” e os “fantasmas”. O jogador controla o personagem “come-come”, que tem o objetivo de comer todas as bolinhas do cenário para vencer o jogo, e o “fantasma” tenta impedi-lo, tentando encurralá-lo. O “come-come” morre quando um “fantasma” consegue encostar-se a ele.

O fantasma tem o objetivo de se dirigir para o local onde está o “come-come”, para isso precisa calcular o caminho mais curto de sua posição até a posição do “come-come”.

O labirinto, cenário principal do jogo PacMan, será um cenário semelhante a de um jogo PacMan, e não o verdadeiro cenário utilizado pelos jogos PacMan. Neste caso, o labirinto, pode ser modelada como um grafo não dirigido, e o planejamento do caminho de cada “fantasma” se resume a um problema de busca por caminho mínimo. Cada quadrado que compõe o cenário do jogo é uma possível posição dos personagens, que pode estar livre ou com obstáculos (quadrado obstruído ou parede). No grafo, os quadrados livres serão vértices.

As arestas ligam quadrados vizinhos entre si. A figura 12 ilustra a modelagem deste grafo.

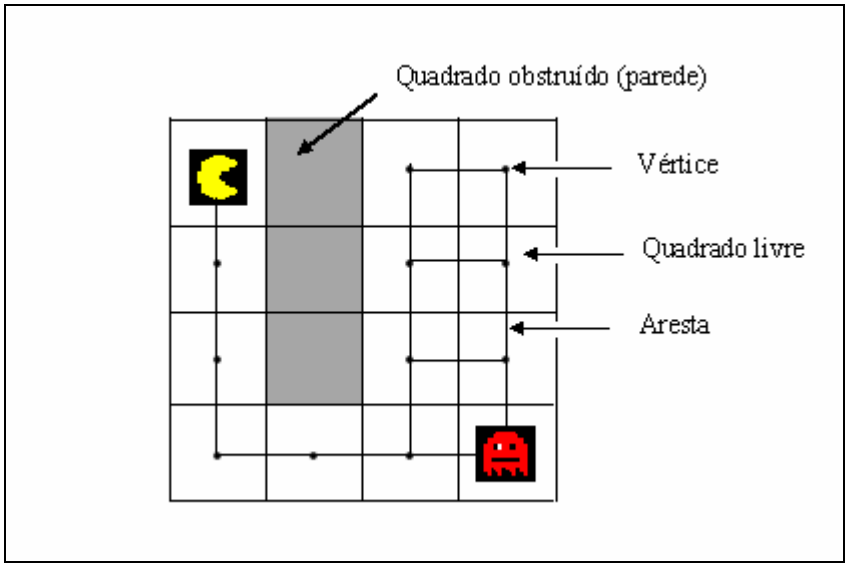


Figura 12 – Modelagem do grafo

Para ajudar o “fantasma” a concluir o seu objetivo é necessária a utilização de métodos de busca. Dentre os métodos de busca será utilizado o busca em largura, o busca em profundidade, o A*, além também do busca em profundidade adaptado e o A* adaptado que são buscas específicas para o problema proposto.

3.1.2 Requisitos funcionais e não funcionais

Os requisitos funcionais são:

a) implementação do algoritmo busca em largura, para planejamento de caminho de personagens de jogos tipo PacMan;
b) implementação do algoritmo busca em profundidade, para planejamento de caminho de personagens de jogos tipo PacMan;
c) implementação do algoritmo busca em profundidade adaptado, para planejamento de caminho de personagens de jogos tipo PacMan;
d) implementação do algoritmo A*, para planejamento de caminho de personagens de

jogos tipo PacMan;

e) implementação do algoritmo A* adaptado, para planejamento de caminho de personagens de jogos tipo PacMan;

f) visualização gráfica do ambiente semelhante ao jogo PacMan, demonstrando o funcionamento da busca em largura, busca em profundidade, busca em profundidade otimizada, algoritmo A* e do algoritmo A* adaptado.

O requisito não funcional é utilizar a linguagem de programação *Object Pascal* com ambiente *Delphi*.

3.2 ESPECIFICAÇÃO

Será apresentada a ferramenta utilizada para a especificação do problema, bem como os diagramas que foram gerados a partir dela, que representam logicamente o trabalho desenvolvido.

3.2.1 Ferramenta utilizada

Para a modelagem do protótipo do software são utilizadas técnicas de orientação a objetos (OO) com *Unified Modelling Language* (UML). Foi utilizada a ferramenta Rational Rose Requisite-Pro para modelar o diagrama de casos de uso, diagrama de classes e os digramas de seqüências.

3.2.2 Diagrama de casos de uso

Nesta seção é mostrado o diagramas de casos de uso, demonstrando os casos de uso implementados para o jogo.

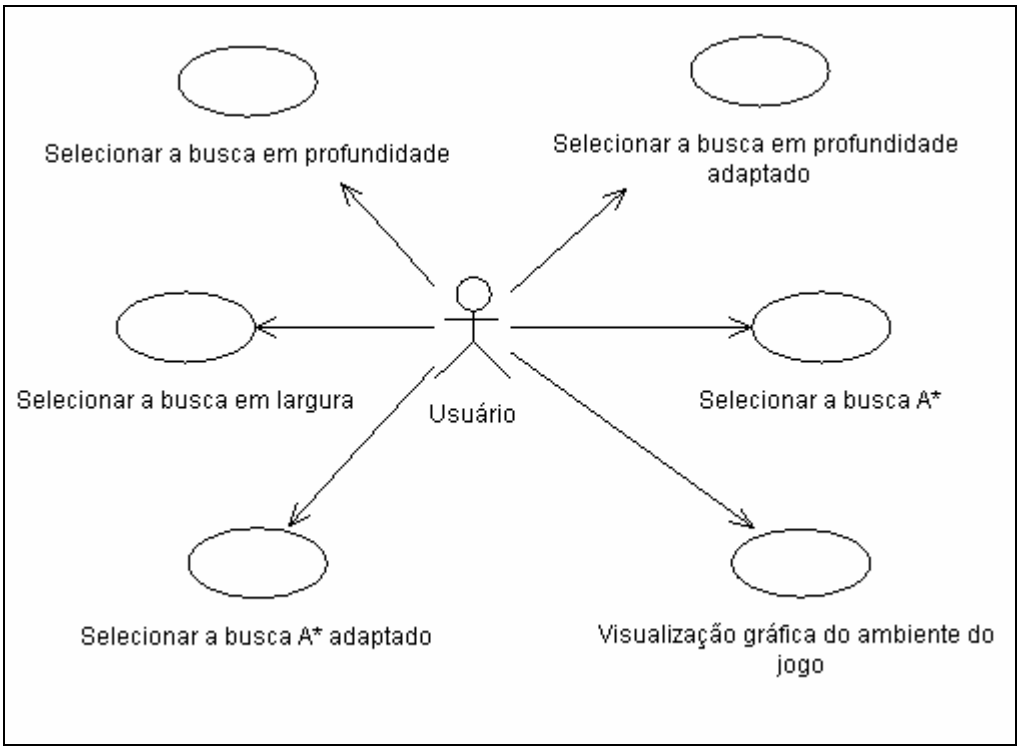


Figura 13 – Diagrama de casos de uso

3.2.3 Diagrama de classes

Nesta seção é mostrado o diagramas de classes, demonstrando as ligações entre as classes e suas multiplicidades.

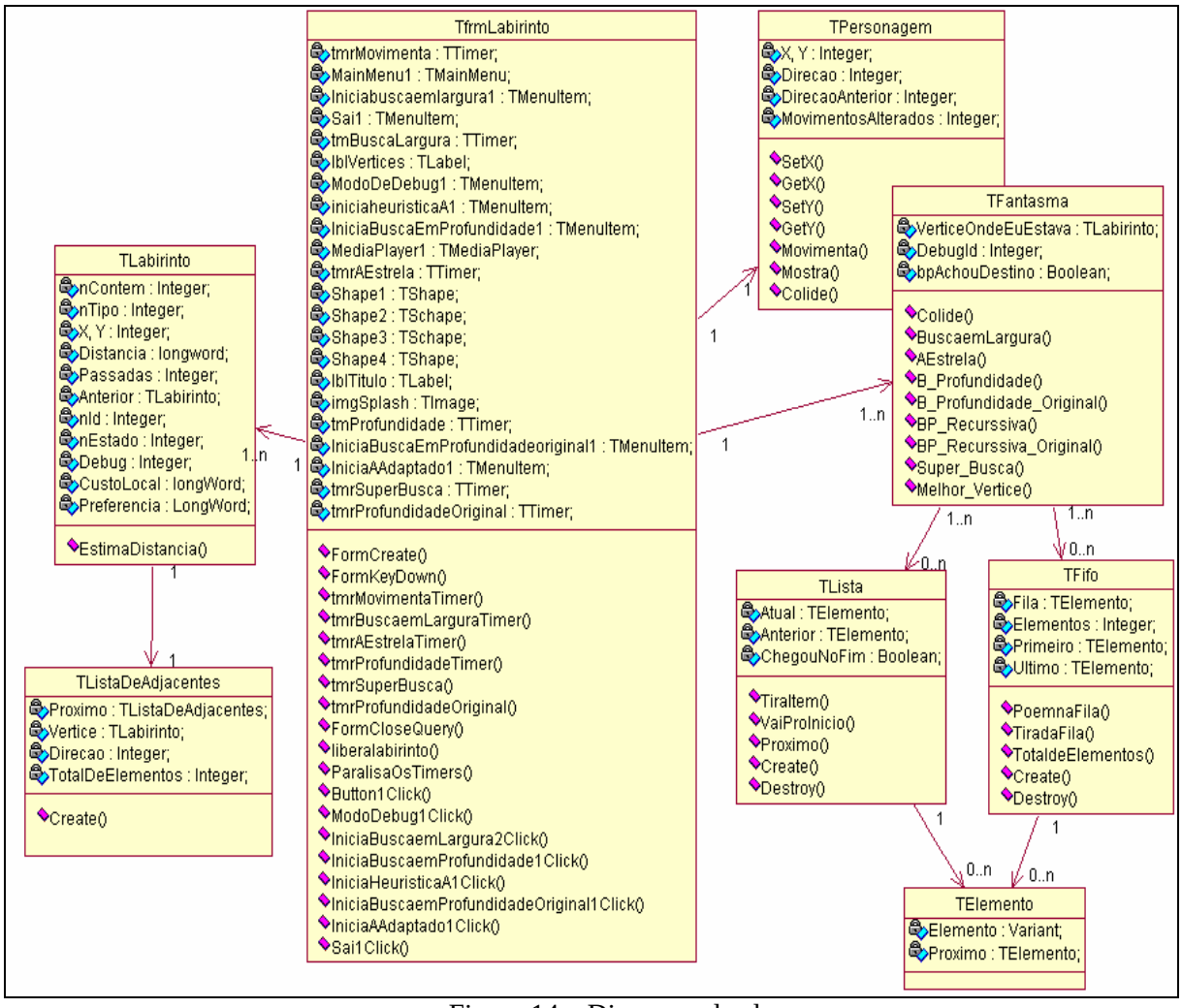


Figura 14 – Diagrama de classes

As classes explicadas abaixo implementam todo o jogo e o seu funcionamento:

a) TfrmLabirinto é uma classe que implementa o formulário principal do jogo, dentro de seus eventos todos os outros objetos são criados e manipulados;

b) TLabirinto é uma classe que implementa um pequeno bloco do labirinto, ou seja, cada objeto criado através desta classe poderá ser considerado um vértice no labirinto. Ela possui todas as funções para ser incorporada ao formulário principal, e se tornar visível na tela;

c) TListaDeAdjacentes é uma classe usada para implementar o grafo. Cada objeto TlistaDeAdjacentes, aponta para um objeto TLabirinto e também para um outro objeto de seu mesmo tipo;

d) TPersonagem é uma classe que instancia apenas um único objeto, e que implementa o personagem principal do jogo, o “come-come”. A classe contém métodos para mover o personagem na tela, e reproduzir as animações;

e) TFantasma é uma classe derivada de TPersonagem, que serve para fazer os “fantasmas”. Re-implementa algumas funções de TPersonagem, e o principal é que todos os algoritmos de busca foram implementados nesta classe.

f) TElemento é uma classe que possui um elemento básico, que pode ser um número ou uma *string* de qualquer tamanho (pois utiliza o tipo de dados *variant*). Utilizada principalmente para encadear as informações. Cada TElemento possui um ponteiro para o próximo TElemento;

g) TFifo é uma classe que implementa uma fila FIFO (*first in first out*), ou seja, onde os primeiros elementos a serem armazenados são também os primeiros elementos a serem retirados, utilizada apenas pelo algoritmo busca em largura;

h) Tlista é uma classe derivada de TFifo, que implementa uma lista (objeto que permite pesquisas aleatórias, e retiradas aleatórias) que são usadas pelos algoritmos busca em profundidade, algoritmo A*, A* adaptado e busca em profundidade adaptado.

3.2.4 Diagramas de seqüências

Nesta seção serão mostrados os diagramas de seqüências da criação do labirinto do jogo PacMan, e das buscas implementadas.

A figura 15 representa a criação do labirinto. O labirinto é criado dinamicamente. A classe TFrmLabirinto, que é a responsável pelo formulário principal do jogo, faz as chamadas para as classes necessárias para o labirinto e os personagens se tornarem visíveis, modelando o labirinto como uma lista de adjacentes.

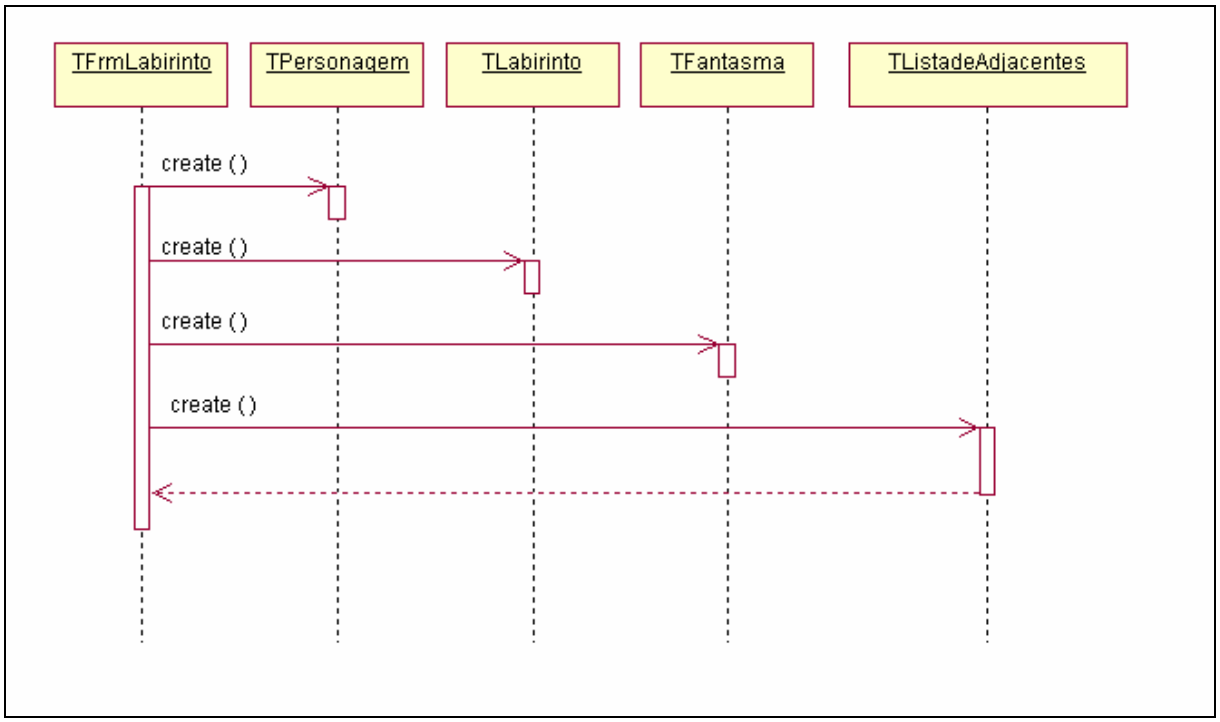


Figura 15 – Diagrama de seqüências da criação do labirinto

A figura 16 representa a busca em profundidade em execução.

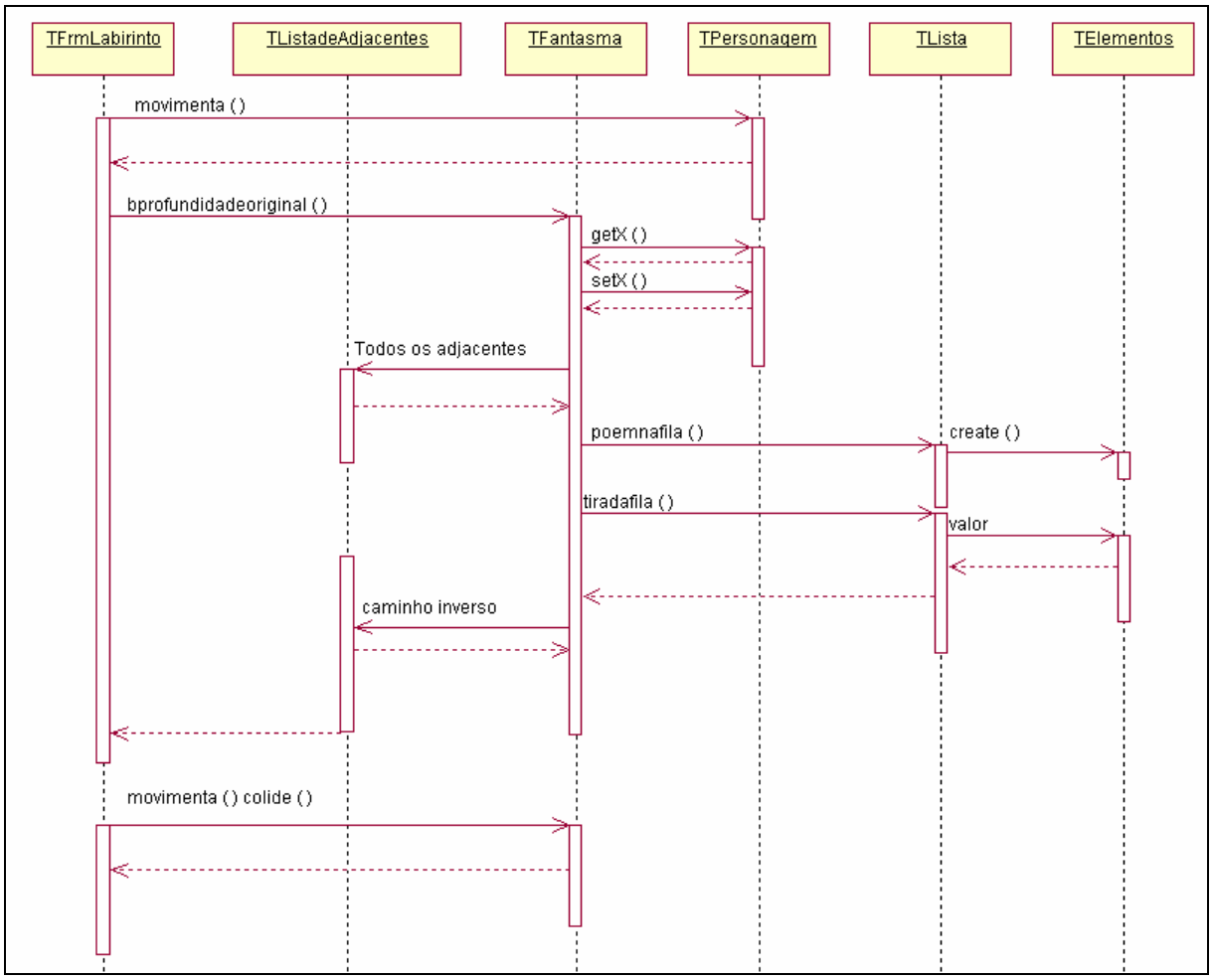


Figura 16 – Diagrama de seqüências da busca em profundidade

A figura 17 representa a busca em largura em execução. A busca em largura e a busca em profundidade são parecidas, afinal a lógica das duas é a mesma, o que muda é que a busca em largura utiliza uma fila e a busca em profundidade uma pilha, neste caso uma lista. A classe TLista herdou os objetos da classe TFifo, por isso é utilizado poemnafila e tiradafila em todos os diagramas, apesar das classes serem diferentes.

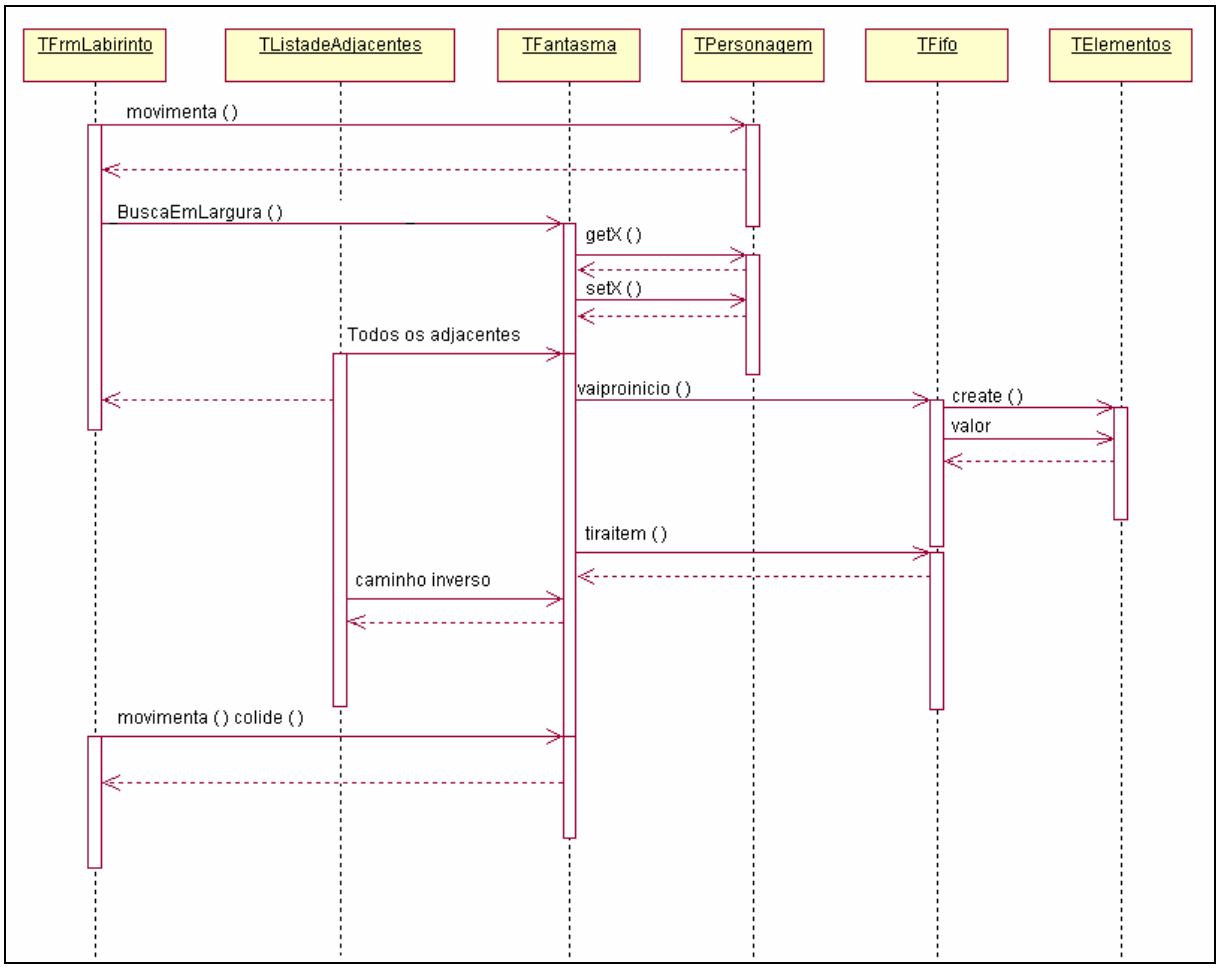


Figura 17 – Diagrama de seqüências da busca em largura

A figura 18 representa a execução do algoritmo A* e do A* adaptado. O diagrama de seqüência para estas duas buscas é o mesmo, pois a única diferença é que o A* adaptado possui um número limitado de vértices que podem ser tirados da lista de abertos, controlado através de um *if*.

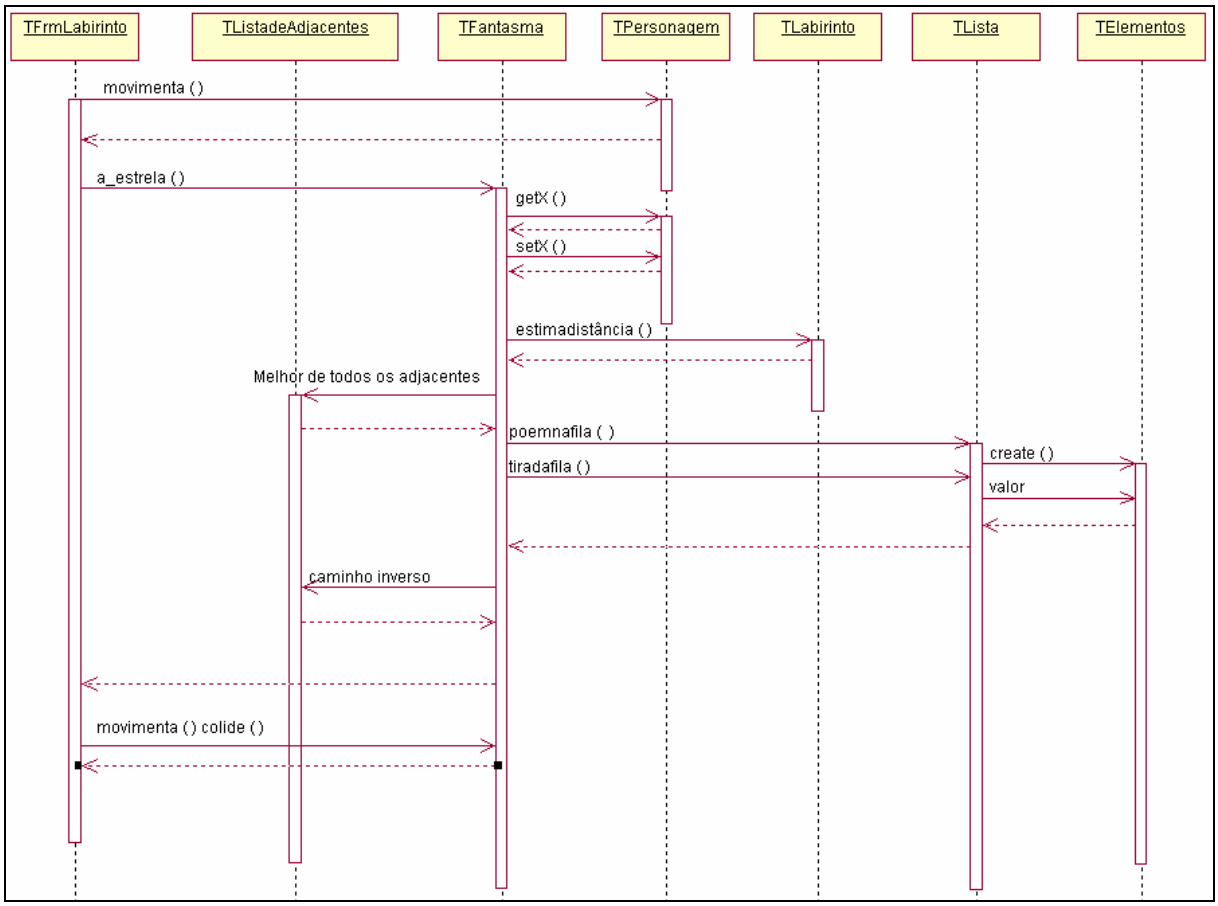


Figura 18 – Diagrama de seqüências da busca heurística A* e A* adaptado

A figura 19 representa a execução do algoritmo busca em profundidade adaptado. Como pode ser visto no diagrama de seqüências ele mescla o busca em profundidade com o A*.

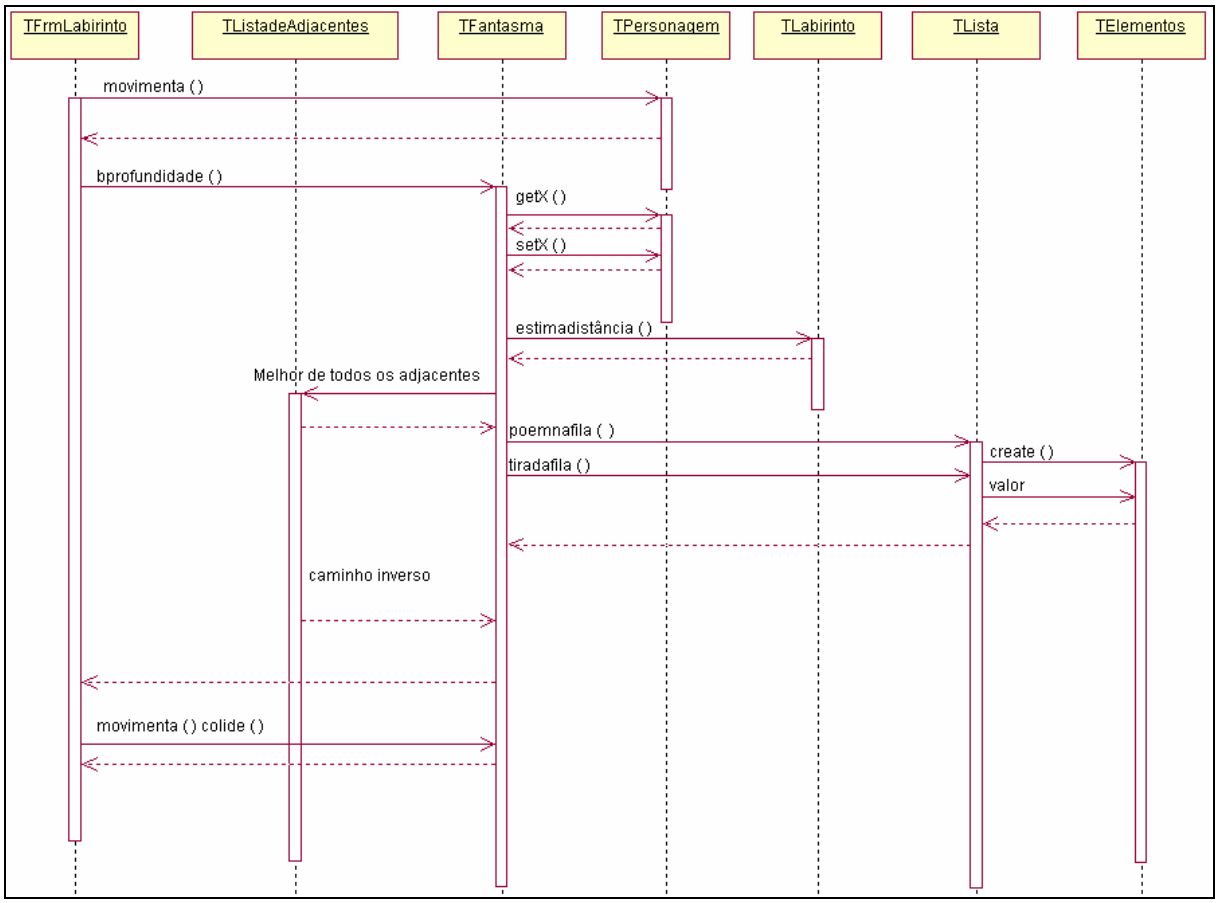


Figura 19 – Diagrama de seqüências da busca em profundidade adaptado

3.3 IMPLEMENTAÇÃO

São apresentadas as técnicas e ferramenta utilizadas, os principais códigos fontes implementados e o funcionamento da implementação em nível do usuário.

3.3.1 Ferramenta utilizada

Foi utilizada a linguagem de programação *Object Pascal* com ambiente Delphi para implementar o ambiente do jogo e os algoritmos de busca.

3.3.2 Técnicas utilizadas

A seguir, é explicado como foi implementado o cenário principal do jogo, ou seja, o labirinto.

3.3.2.1 O labirinto

O cenário inicial do jogo é implementado utilizando de um classe TfrmLabirinto. O TfrmLabirinto.FormCreate, é o método onde as variáveis globais são inicializadas e as propriedades de objetos automáticos são definidos. O labirinto é criado através de texturas carregadas dinamicamente, sendo assim o grafo é definido.

O desenho que é usado para desenhar o labirinto está definido na variável Labirinto que é apenas uma *string*, que define onde está à parede, o “fantasma”, e o “come-come”.

O labirinto foi modelado com 20 blocos de largura por 20 blocos de altura. Estes blocos são colocados em uma matriz chamada Lab, que foi declarada globalmente da seguinte forma: *Lab : array[1..altura] of array[1..largura] of TLabirinto*, ou seja, uma matriz de 20 por 20, contendo objetos da classe TLabirinto, que representam um pequeno bloco. De acordo com as instruções da matriz Lab, as propriedades destes objetos são alteradas, para que o bloco seja um espaço por onde se possa caminhar ou pode ter uma parede (obstáculo), que pode conter comida ou uma vitamina. O quadro 4 mostra a implementação da matriz Lab.

Quando encontra uma instrução "C" ou "F" para o PacMan e os Fantasmas respectivamente, criam-se os objetos correspondentes. Para os Fantasmas, a cada nova instrução "F" encontrada na matriz Lab, uma matriz de fantasmas é ampliada e um novo fantasma é criado.

Ao mesmo tempo em que o labirinto vai sendo criado é também desenhado na tela.

Após este processo, já se sabe exatamente pelas propriedades de cada item de LAB, o que é um espaço caminhável, e o que é parede. Cada bloco de labirinto por onde o personagem pode caminhar é considerado um vértice. Então é o momento de criar o grafo que será usado por todos os algoritmos de busca.

O grafo é criado a partir de dois laços *for*, utilizados para passar por todos os vértices do labirinto LAB, tudo isso para alimentar um vetor nomeado de grafo, que tem o número de elementos igual ao número de vértices criados para o labirinto. Então têm-se algo assim: Grafo[0] recebe o Primeiro vértice , Grafo[1] recebe o segundo vértice e assim sucessivamente. Porém somente isso não monta o grafo, pois o primeiro vértice é um objeto de TListadeAdjacentes, que é usada para implementar o grafo, e tem um ponteiro apontando para LAB[x,y] e outro ponteiro apontando para o próximo TlistaDeAdjacentes que contém um vértice próximo. O primeiro vértice aponta para o vértice logo abaixo, que aponta para o vértice à sua esquerda, que aponta para o vértice à direita, que aponta para o vértice acima.

É claro que se o bloco logo acima, ou à direita ou qualquer um, for uma parede, ele não será incluso, o que gera um número bem variável de vértices adjacentes para cada vértice raiz. Desta forma é criada a lista de adjacências que constitui o grafo.

```
//Inicia a string que será o labirinto
          // P = Parede
          // V = Vitamina
          // C = PacMan
          // F = Fantasma
          //' '= Espaço livre com comida

     Const Labirinto: AnsiString = 'PPPPPPPPPPPPPPPPPPP' +
                                   'PV                VP' +
                                   'P PPP PPP P PP PPP P' +
                                   'P P     P P        P' +
                                   'P PPP P P P PP PPP P' +
                                   'P       P          P' +
                                   'P PPP PPP PPPP PPP P' +
                                   'P P       P      P P' +
                                   'P PPP P P P PP PPP P' +
                                   'P                  P' +
                                   'P PPP P P P PP PPP P' +
                                   'P P F   P P      PCP' +
                                   'P P PPPPP PPPPPP P P' +
                                   'P       P P        P' +
                                   'P PPP P P P PP PPP P' +
                                   'P P     P P      P P' +
                                   'P P PPPPP PPPPPP P P' +
                                   'P p ppppp pppppp   P' +
                                   'PV                VP' +
                                   'PPPPPPPPPPPPPPPPPPP' ;
```

Quadro 4 – Código fonte da matriz LAB

Os códigos fonte em *Object Pascal* dos algoritmos de busca implementados para o jogo serão mostrados na seção Apêndice.

3.3.3 Operacionalidade da implementação

Serão apresentadas as telas do programa, e a explicação de cada tela.

3.3.3.1 Telas do programa

A figura 20 é a tela inicial do cenário do jogo PacMan. O labirinto é onde o jogo PacMan acontece e onde as buscas são realizadas. Ao lado do labirinto é avaliado o progresso da busca em execução, através dos: vértices testados pela busca, média de vértices por

movimento, total de vértices testados e o total de movimentos, sendo que a avaliação é atualizada a cada novo movimento do personagem “come-come”.

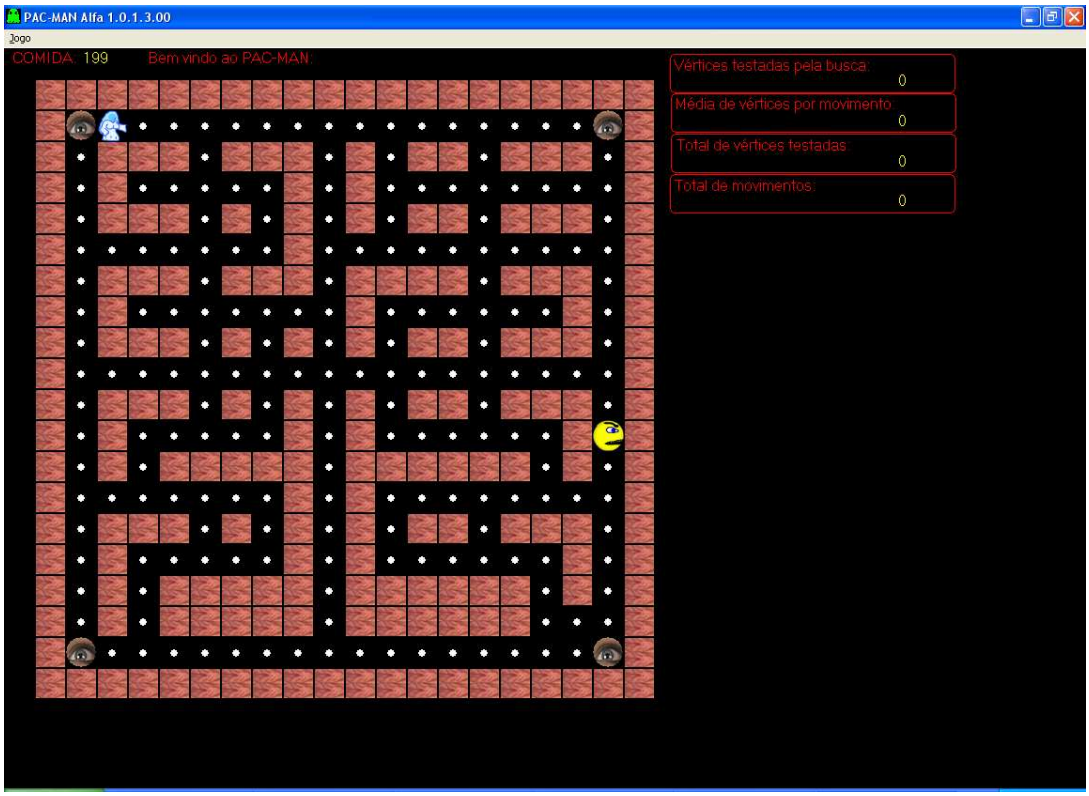


Figura 20 – Cenário inicial do jogo

A figura 21 é a tela do cenário do jogo com o menu de opções jogo. Através dele, o jogo é inicializado. Dentre as opções disponíveis do menu jogo, existem o modo *debug*, a busca em largura, a busca em profundidade, a busca em profundidade adaptado, a busca heurística A* e o A* adaptado. Pode ser selecionada uma das buscas disponíveis para ver o jogo em funcionamento do modo tradicional do jogo PacMan, ou seja, o personagem “come-come” comendo as bolinhas do labirinto, ou pode-se escolher o modo *debug* antes de selecionar uma das buscas, este modo deixará visível o funcionamento da busca em execução. As bolinhas em vermelho são os vértices que estão sendo visitados pelo busca em execução, e as bolinhas em amarelo demonstram o caminho mínimo do personagem “fantasma” até o personagem “come-come”.

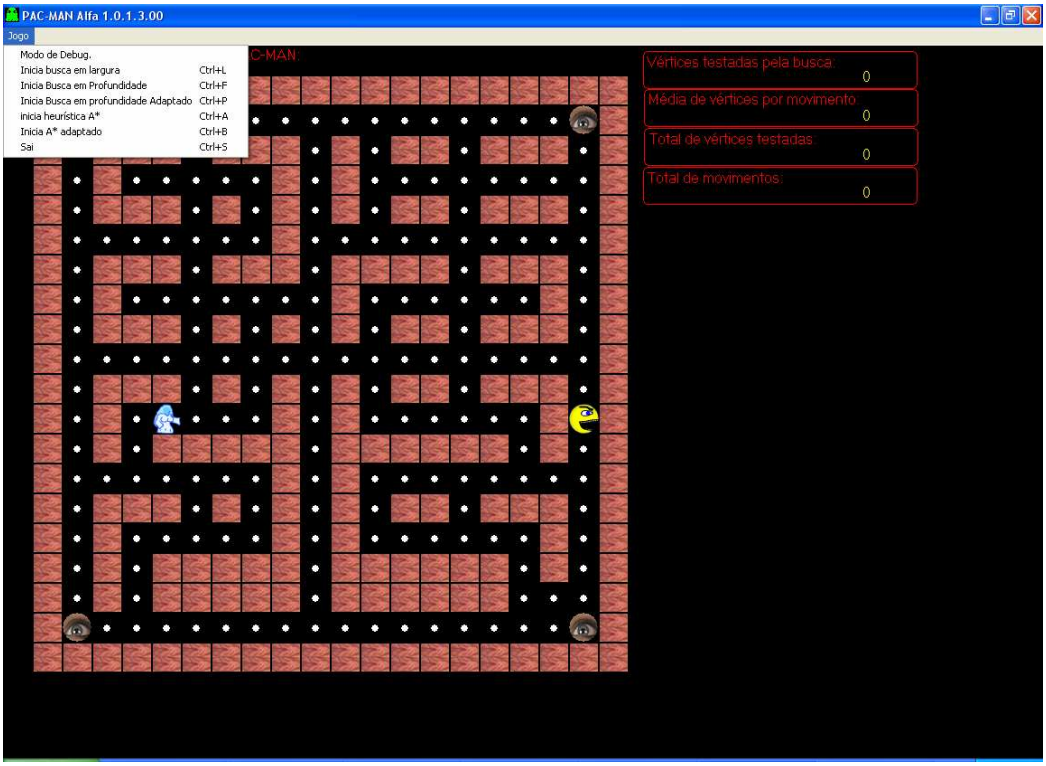


Figura 21 – Cenário inicial do jogo com o menu de opções

Na figura 22 é demonstrado o usuário jogando contra o busca em largura.

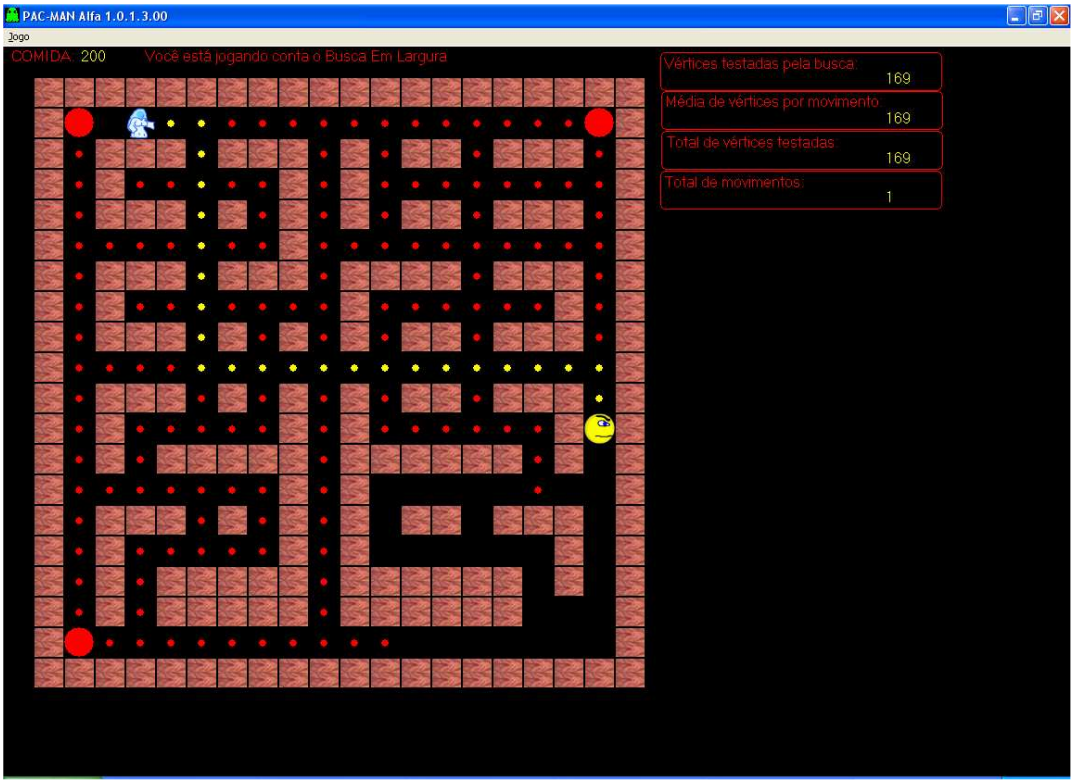


Figura 22 – Usuário jogando contra o busca em largura

Na figura 23 é demonstrado o usuário jogando contra o busca em profundidade.

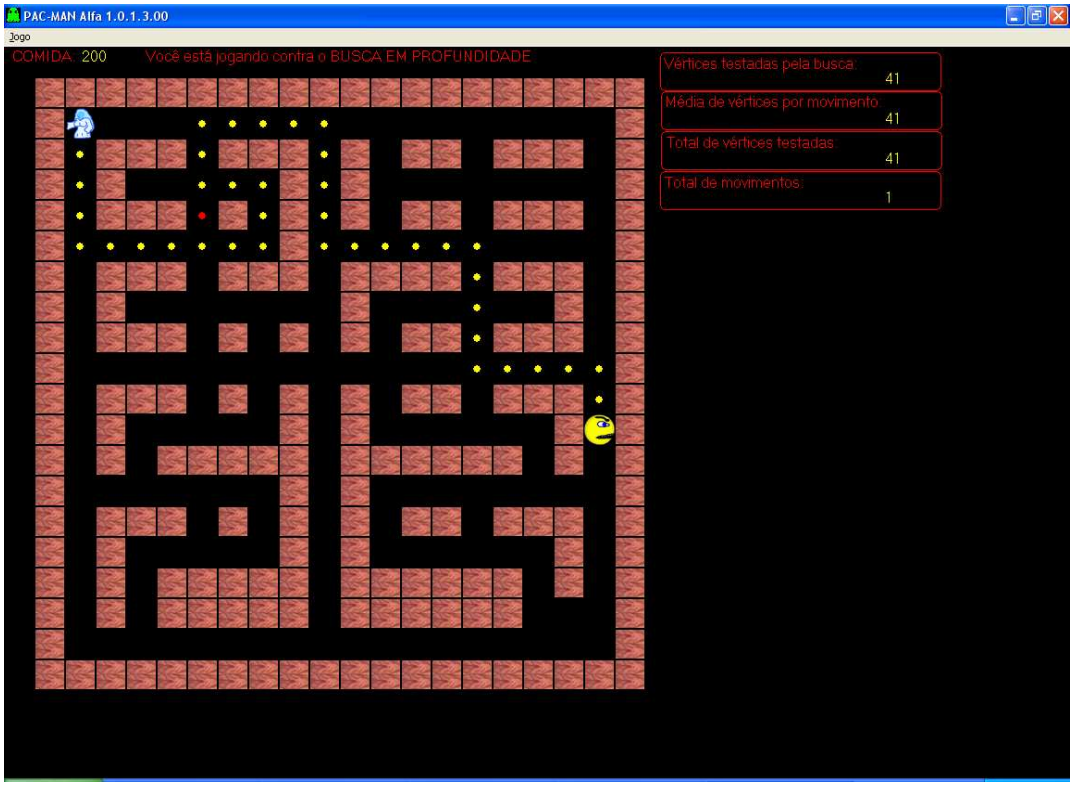


Figura 23 – Usuário jogando contra o busca em profundidade

Na figura 24 é demonstrado o usuário jogando contra o busca em profundidade adaptado.

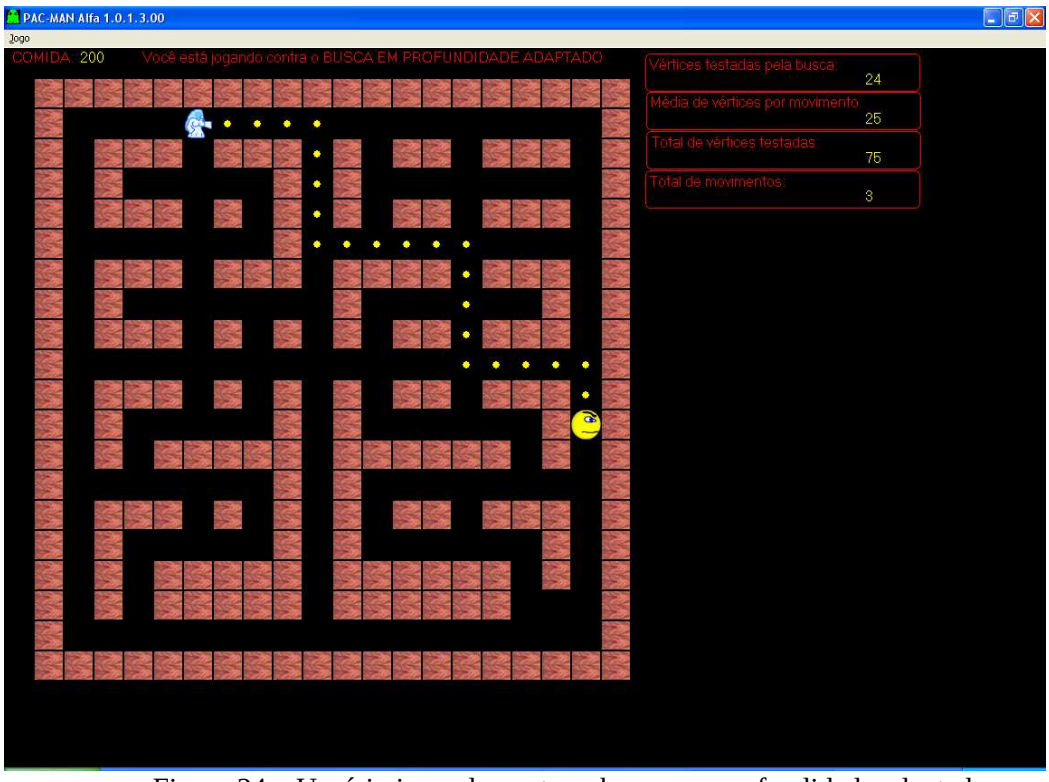


Figura 24 – Usuário jogando contra o busca em profundidade adaptado

Na figura 25 é demonstrado o usuário jogando contra o A*.

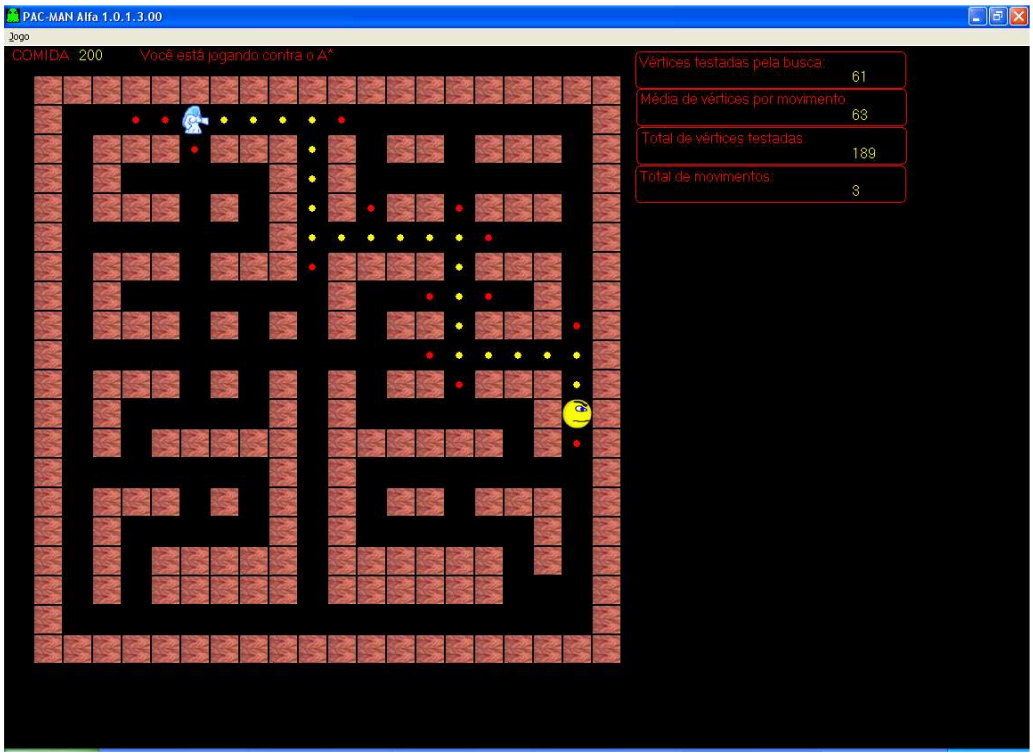


Figura 25 – Usuário jogando contra o busca heurística A*

Na figura 26 é demonstrado o usuário jogando contra o A* adaptado. “come-come”.

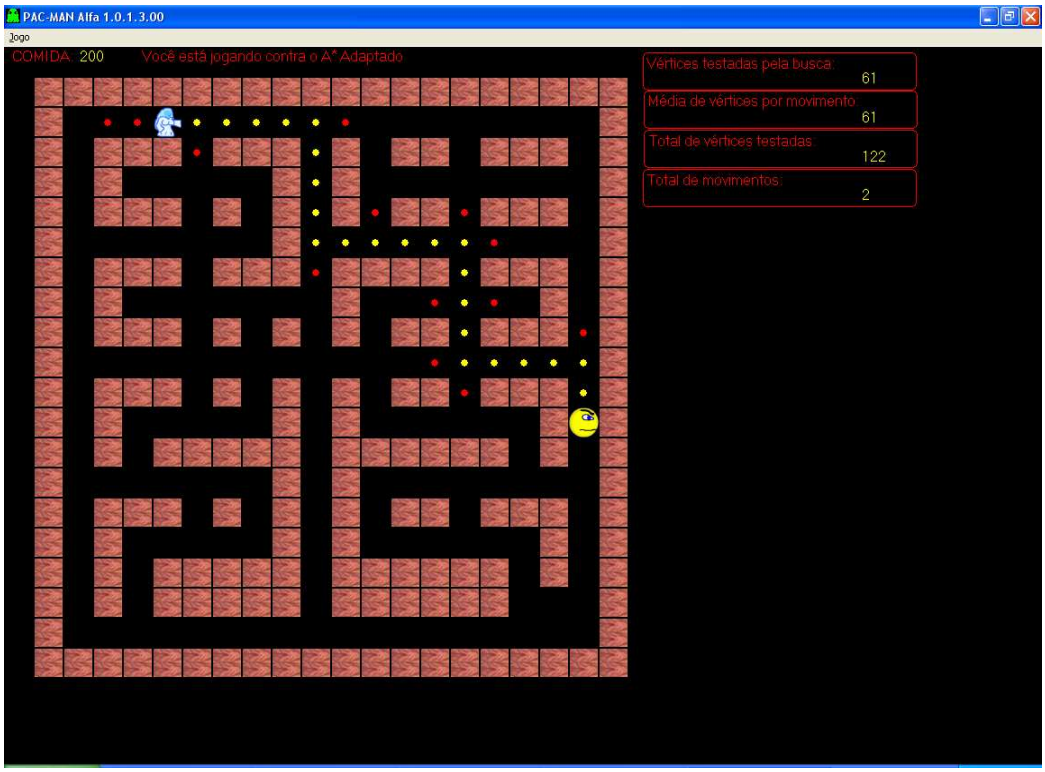


Figura 26 – Usuário jogando contra o A* adaptado

3.4 RESULTADOS E DISCUSSÃO

Foram realizados testes para verificar o desempenho dos algoritmos implementados. Serão mostrados os resultados de cada método de busca, e um comparativo entre os resultados.

Os testes consistem em jogar contra cada um dos algoritmos de busca, colocar o personagem “come-come” na mesma posição para comparar o desempenho entre as buscas, e anotar o seu desempenho: vértices testados pela busca (é o número de vértices testados a cada novo movimento dos personagens), a média de vértices por movimento (é a soma do total de movimentos dividido pelo total de vértices testados), o total de vértices testados e o total de movimentos. Para cada algoritmo foram realizados cinco testes, com exceção do busca em profundidade que será mostrado somente um.

3.4.1 Busca em largura

Na tabela 1 é mostrado os resultados dos testes realizados com o busca em largura.

Vértices testados pela busca	Média de vértices por movimento	Total de vértices testados	Total de movimentos
5	116	2.890	25
7	129	4.108	32
4	79	1.172	15
6	123	3.793	31
5	107	3.312	31

Tabela 1 – Resultados dos testes realizados com o busca em largura

3.4.2 Busca em profundidade

Na tabela 2 é demonstrado o desempenho da busca em profundidade. O algoritmo de

busca em profundidade, no caso do jogo PacMan, tem uma particularidade: o “fantasma”, personagem onde o algoritmo é aplicado, encontra um caminho até o “come-come”, a cada novo movimento do fantasma o algoritmo encontra um novo caminho, perdendo-se pelo labirinto e não chegando ao “come-come”, motivo pelo qual não serão executados 10 testes.

No primeiro teste, o fantasma não obteve êxito, testou milhares de vértices sem sucesso, a ponto da busca ter que ser interrompida, pois o fantasma nunca chegaria até o come-come.

No segundo teste (tabela 2), o personagem “come-come” foi levado até uma posição onde o “fantasma” não poderia mudar o seu percurso e encontrando assim o “come-come”.

Vértices testados pela busca	Média de vértices por movimento	Total de vértices testados	Total de movimentos
2	31	17.277	560

Tabela 2 – Resultado do teste realizado com o busca em profundidade

3.4.3 Busca em profundidade adaptado

Na tabela 3 é mostrado os resultados dos testes realizados com o busca em profundidade adaptado.

Vértices testados pela busca	Média de vértices por movimento	Total de vértices testados	Total de movimentos
2	14	350	25
2	18	557	32
2	13	189	15
2	15	462	31
2	16	484	31

Tabela 3 – Resultados dos testes realizados com o busca em profundidade adaptado

3.4.4 Busca heurística A*

Na tabela 4 é mostrado os resultados dos testes realizados com o busca heurística A*.

Vértices testados pela busca	Média de vértices por movimento	Total de vértices testados	Total de movimentos
7	38	927	25
8	40	1.271	32
6	31	462	15
8	40	1.238	31
7	37	1.137	31

Tabela 4 – Resultados dos testes realizados com o busca heurística A*

3.4.5 A* adaptado

Na tabela 5 é mostrado os resultados dos testes realizados com o A* adaptado.

Vértices testados pela busca	Média de vértices por movimento	Total de vértices testados	Total de movimentos
7	37	921	25
8	41	1.290	32
6	31	454	15
8	40	1.216	31
7	36	1.113	31

Tabela 5 – Resultados dos testes realizados com o A* adaptado

3.4.6 Comparativo entre os resultados das buscas

Na tabela 6 são mostradas os resultados obtidos com os testes levando em consideração a média de vértices testados por cada método de busca.

Busca em largura	Busca em profundidade adaptado	Busca heurística A*	A* adaptado
116	14	38	37
129	18	40	41
79	13	31	31
123	15	40	40
107	16	37	36

Tabela 6 – Comparativo entre as buscas

3.4.7 Comparando os resultados obtidos com os trabalhos correlatos

De acordo com os resultados obtidos com os algoritmos de busca implementados em comparativo com os trabalhos de Santos (2002), Lester (2003), Matthews (2000) e Pereira (1999) diante do desempenho de cada algoritmo de busca genérico, o algoritmo A* encontra sempre o menor caminho entre dois vértices em um grafo e faz isso relativamente rápido.

4 CONCLUSÕES

Diante do objetivo deste trabalho, fazer um estudo comparativo entre os algoritmos de busca de caminhos, o busca em largura, o busca em profundidade e o algoritmo A*, entre os personagens do jogo PacMan, os objetivos foram alcançados e superados, pois além do estudo comparativo entre os algoritmos mencionados, foram implementados também os algoritmos de busca em profundidade adaptado e o A* adaptado, sendo que estes algoritmos são específicos para o problema proposto, o jogo PacMan.

O algoritmo de busca em largura, em relação aos outros algoritmos implementados, encontra a solução do problema, porém testa uma grande quantidade de vértices, mesmo sabendo onde está o vértice que ele está procurando, mesmo assim ele é melhor que o busca em profundidade pois acha uma solução.

O algoritmo busca em profundidade, conforme os testes realizados, não é eficiente para o problema proposto, pois o “fantasma” dificilmente chega até onde está o “come-come”.

O algoritmo A* encontra sempre uma boa solução, ou seja, um caminho sem ter que testar uma grande quantidade de vértices, levando em consideração que é um algoritmo genérico, e que não foi criado para ser utilizado exclusivamente neste jogo.

O algoritmo A* adaptado, é a implementação do algoritmo A* com número limitado de vértices que podem ser tirados da lista de abertos, o número escolhido foi 25 vértices, levando em consideração o tamanho do labirinto implementado. O A* adaptado, às vezes testa menos vértices que o A*, porém na maioria dos testes realizados ele deixa de testar um vértice importante para a solução do problema, acabando testando mais vértices que o A*.

O algoritmo busca em profundidade adaptado mescla o algoritmo busca em profundidade com o algoritmo A*. Diante do problema proposto ele foi o algoritmo com o melhor desempenho, ou seja, a menor média de vértices testados. O busca em profundidade

ordena os vértices adjacentes a serem testados por ordem de menor custo, colocando todos os adjacentes não testados em uma lista, calcula a distância, que é estimada conforme a fórmula do A* e adiciona um custo local. O custo local do vértice aumenta cada vez que um "fantasma" passa por ele, e é zerado quando o "come-come" passa pelo mesmo vértice, isso desestimula o "fantasma" a andar em círculos, (que é o problema do busca em profundidade implementado) e os estimula a seguir o "come-come". Entretanto este é um algoritmo que foi implementado para o jogo PacMan e não pode ser usado para a solução de problemas genéricos de busca.

O uso da linguagem de programação *Object Pascal* com ambiente *Delphi* para a realização deste estudo comparativo foi eficiente, pois ajudou a alcançar os objetivos propostos. Em relação a modelagem de objetos para implementação do protótipo, a separação entre classes de modelo e interface não foi proposta como requisito porque o principal objetivo da implementação era a criação de um protótipo de software com o qual se poderia comparar os diferentes algoritmos de busca em uma aplicação semelhante a um jogo, ao invés de implementar um jogo propriamente dito.

Dentre as possibilidades de continuidade do trabalho realizado, poderia ser feita uma adaptação do algoritmo busca em largura com o algoritmo A*, transformar o cenário do jogo (o labirinto) em 3D e fazer um estudo de outros algoritmos para resolução de problemas de busca de caminho mínimo para aplicá-los no jogo implementado.

REFERÊNCIAS BIBLIOGRÁFICAS

ADAMS, Douglas. **The hichhiker´s guide to the galaxy.** Londres. 2003. Disponível em: <http://www.douglasadams.com/creations/infocomjava.html>. Acesso em: 15 maio 2005.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 6023**: informação e documentação: referências - elaboração. Rio de Janeiro, 2002a.

_____. **NBR 6024**: informação e documentação: numeração progressiva das seções de um documento escrito - apresentação. Rio de Janeiro, 2003a.

_____. **NBR 6027**: informação e documentação: sumário - apresentação. Rio de Janeiro, 2003b.

_____. **NBR 6028**: resumos. Rio de Janeiro, 2003c.

BATTAIOLA, André L. Jogos por computador. In: CONGRESSO NACIONAL DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO. 19., 2000, Curitiba. **Anais do Congresso Nacional da Sociedade Brasileira de Computação**. Curitiba: PUCPR, 2000. p. 83 – 122.

CORMEM, Thomas H. et al. **Algoritmos**: teoria e prática. 2. ed. Rio de Janeiro: Editora Campus, 2002.

CRUDO, Ricardo de L. **IAdvanced lessons & dragons:** aspectos benéficos do RPG, MUD e jogos computacionais. 2001.65 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Departamento de Ciência da Computação, Universidade Federal de Cuiabá, Cuiabá. Disponível em: < http://64.233.187.104/search?q=cache:mEgGaYIFzKwJ:www.ufmt.br/cacomp/Downloads/monografias/advancedLessonsAndDragons.pdf+arcades+aventura+estrategias+guerra+figura+jogo&hl=pt-BR&lr=lang_pt>. Acesso em: 10 maio 2005.

LESTER, Patrick. **A* pathfinding for beginners**. [S.l], 2003. Disponível em: <http://www.generation5.org/content/2000/astar.asp> Acesso em: 13 fev. 2005.

MATTHEWS, James. **A* for the masses**. [S.l], 2000. Disponível em: <http://www.generation5.org/content/2000/astar.asp>. Acesso em: 24 mar. 2005.

MELO, Ana C. **Desenvolvendo aplicações com UML**. Rio de Janeiro: Brasport, 2002.

PEREIRA, Charles**. Implementação de heurística para determinação do caminho de menor custo**. 1999. 57 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

POSTMA, Brian. **PacMan**. [S.l], 2000. Disponível em: <http://www.homepages.hetnet.nl/~brianpostma/pacman.html>. Acesso em: 14 maio 2005.

RABIN, Steven. **AI game programming wisdow**. Massachusets: Charles River Media, Inc., 2002.

RABUSKE, Renato A . **Inteligência artificial**. Florianópolis: Editora da UFSC, 1995.

RICH, Elaine; KNIGHT, Kevin. **Inteligência artificial**. São Paulo: Makron Books, 1993.

ROLLINGS, Andrew; ADAMS, Ernest**. Andrew Rollings and Ernest Adams on game design**. Indianapolis: New Riders, 2003.

RUSSELL, Stuart; NORVIG, Peter. **Artificial intelligence**: a modern approach. New Jersey: Prentice Hall, 2003.

SANTOS, Gilliard L. **Inteligência artificial em jogos 3D**. 2002. 39 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Departamento de Ciência da Computação, Universidade Federal Fluminense, Niterói. Disponível em: <http://www.paralelo.com.br/documentos/Real-Time%20BSP-Based%20Path%20Planning%20with%20A-Star%20(Full-Portuguese).pdf>. Acesso em: 05 mar. 2005.

Wikipedia. [S.l], 2000. Disponível em: <http://pt.wikipedia.org/wiki/Jogo_de_computador>. Acesso em: 27 mar. 2005.

APÊNDICE A – Códigos fontes

Serão mostrados os códigos fontes em *Object Pascal* dos algoritmos de busca implementados.

O quadro 5 traz o código fonte da implementação do algoritmo de busca em largura.

```
Procedure TFantasma.BuscaEmLargura(DestinoX:Integer; DestinoY :Integer);
 var
      ultimoAdjunto, nRaizId, tmpId, nId : Integer;
      nDirecao :Integer; //Nova direção do fantasma
      Fila : TFifo;
      PLab : TLabirinto; //Para receber o retorno da fila
      X,TotalDeAdjacentes : Integer;
      PGrafo : TListaDeAdjacentes;
      SaiAgora : Boolean;
      nContaVertices : Integer;//Conta o nº de vértices testados
begin
      //Descobre o Id raiz
      nRaizId := lab[self.X,self.Y].nID;
      //Zera o indicador de passadas.E a cor real do labirinto
      for X:=0 to NumeroDeVertices -1 do begin
        grafo[x].Vertice.Passadas:=0;
          if MODO_DEBUG then
            if (grafo[x].vertice.Debug = self.DebugId) or
(grafo[x].vertice.Debug = 0) then
            begin
              grafo[x].Vertice.Brush.Color:=clBlack;
              grafo[x].Vertice.Debug := 0;
            end;
          end;
     //Cria a fila
     Fila := TFifo.Create();
      //Coloca o vertice de origem na fila
      Fila.PoemNaFila(nRaizId);
      //Define a distância do vértice raiz como zero
      grafo[nRaizId].Vertice.Distancia := 0;
      grafo[nRaizId].Vertice.Passadas := 1;
      while (Fila.TotalDeElementos()>0) and (SaiAgora=false) do
      begin
        nId := fila.TiraDaFila();
        //Descobre os adjacentes
        TotalDeAdjacentes := grafo[nId].TotalDeElementos;
        PGrafo := Grafo[nId];
        for X :=0 to TotalDeAdjacentes do begin
          if PGrafo.Vertice.Passadas = 0 then
          begin
            PGrafo.Vertice.Passadas := 1;
            //Incrementa a distância
PGrafo.Vertice.Distancia:=grafo[nId].Vertice.Distancia + 1;
            //Guarda o vértice anterior (para procurar o menor caminho)
            PGrafo.Vertice.Anterior := grafo[nId].Vertice;
            //Coloca na fila o vértice
            fila.PoemNaFila( PGrafo.Vertice.nID );
            //Pinta de vermelho a área testada pela busca em largura
            if MODO_DEBUG then begin
```

```
              if (Pgrafo.Vertice.Debug = self.DebugId) or
(Pgrafo.Vertice.Debug = 0) then
              begin
                Pgrafo.vertice.Brush.Color := clred;
                Pgrafo.Vertice.Debug:=self.DebugId;
              end;
            end;
            nContaVertices := nContaVertices + 1;
          end;
           if PGrafo.Vertice.nID = lab[DestinoX,DestinoY].nID then
           begin
             SaiAgora := true;
           end;
           //Avança para o próximo adjacente
             PGrafo := PGrafo.Proximo;
           end;
        //Marca o vértice analisado como vermelho
        grafo[nId].Vertice.Passadas := 2;
        end;
        //Para fazer a média de movimentos
        TotalArestasBL := TotalArestasBL + nContaVertices;
        //Terminado esse processo, o melhor caminho é o caminho inverso
(Vertice.Anterior) até a raiz
        //Descobre o Id do destino
        PLab := lab[DestinoX,DestinoY];
        nId := lab[self.X, self.Y].nID;
        repeat
          PLab := Plab.Anterior;
          //Pinta de Amarelo o caminho mais curto descoberto
          if MODO_DEBUG then
          begin
            PLab.Brush.Color := clyellow;
            PLab.Debug:=self.DebugId;
          end;
          //Descobre a direção a ser tomada pelo fantasma
          nDirecao := plab.nID;
        until (Plab.Anterior.nID = nId) or (Plab.Distancia=0);
          pGrafo := grafo[nDirecao];
          //Descobre a direção a tomar
          while pGrafo.Vertice.nID <> nId do
          begin
            PGrafo := PGrafo.Proximo;
          end;
      str(nContaVertices,S);
      frmLabirinto.lblVertices.Caption:=S;
      self.Direcao:=Pgrafo.Direcao ;
      fila.Free();
end;
```

Quadro 5 – Código fonte do algoritmo de busca em largura

O quadro 6 traz o código fonte da implementação do algoritmo de busca em profundidade.

```
PROCEDURE TFantasma.B_PROFUNDIDADE_ORIGINAL(DestinoX:Integer;
DestinoY:Integer);
var
  X, nRaizId, nDestinoId :Integer;
  nId : Integer;
  nDirecao :Integer; //NovaDireção do fantasma
```

```
  PLab : TLabirinto; //para receber o retorno da fila
  PGrafo : TListaDeAdjacentes;
begin
   //Zera o indicador de passadas
   TotalDeTestestados := 0;
   for X:=0 to NumeroDeVertices -1 do
   begin
     grafo[x].Vertice.Passadas:=0;
     if MODO_DEBUG then
       if (grafo[x].vertice.Debug = self.DebugId) or
(grafo[x].vertice.Debug = 0) then
       begin
         grafo[x].Vertice.Brush.Color:=clBlack;
         grafo[x].Vertice.Debug := 0;
       end;
   end;
   //O algoritmo de busca em profundidade passa por todos os vértices
existentes.
   //Inicia a variável de finalização
   bpAchouDestino := false;
   //Evita que o fantasma volte para o vertice onde estava antes
   self.VerticeOndeEuEstava.Passadas:=1;
   self.VerticeOndeEuEstava:=lab[self.X,self.Y];
   //Descobre o Id raiz
   nRaizId := lab[self.X,self.Y].nID;
   //Descobre o Id de Destino
   nDestinoId := lab[DestinoX,DestinoY].nID ;
   //Chama a busca recurssíva apenas no ponto RAIZ, ao invés de fazer uma a
uma pelo labirinto
   self.BP_Recurssiva_ORIGINAL(nRaizId, nDestinoId);
   //Atualiza as estatísticas
   TotalArestasAE := TotalArestasAE + TotalDeTestestados;
   //Terminado o processo, verifica se encontrou o destino e descobre o
caminho
   if bpAchouDestino then
   begin
     //Descobre o Id do destino
     PLab := lab[DestinoX,DestinoY];
     if TotalDeTestestados > 1 then
     begin
       repeat
         PLab := Plab.Anterior;
         //Pinta de amarelo o caminho mais curto descoberto
           if MODO_DEBUG then
           begin
             PLab.Brush.Color := clyellow;
             PLab.Debug:=self.DebugId;
           end;
           //Se o vértice foi pintado de amarelo na primeira
           //busca, ele faz parte do primeiro caminho
           //encontrado, logo tem preferência a ser testado na
           //próxima busca
           PLab.Preferencia:=Plab.Preferencia + 1;
           //Descobre o vértice por onde o fantasma deverá
           //passar
           nDirecao := plab.nID;
           until (Plab.anterior.nID = nRaizId);
     end
     else
     begin
       nDirecao := plab.nID;
```

```
     end;
     pGrafo := grafo[nDirecao];
     //Descobre a direção a tomar...
     while pGrafo.Vertice.nID <> nRaizId do
     begin
       PGrafo := PGrafo.Proximo;
     end;
     str(TotalDeTestestados,S);
     frmLabirinto.lblVertices.Caption:=S;
     self.Direcao:=Pgrafo.Direcao ;
   end;
end;

procedure TFantasma.BP_Recurssiva_ORIGINAL(ProcuraID :integer; DestinoId:
Integer);
var
  x , nID: Integer;
  PGrafo : TListaDeAdjacentes;
  ListaParaOrdenar : TLista;
Begin
  //Indica que a pesquisa já passou por esse vértice
  grafo[ProcuraID].Vertice.Passadas := 1;
  //Agora verifica todos os adjacentes
  PGrafo := grafo[ProcuraID].Proximo;
  //Testa se o destino não foi encontrado
  if ProcuraId = DestinoId then
    bpAchouDestino := true;
        //Quando encontra o destino, nada mais deve ser feito
        if not bpAchouDestino then
        begin
          ListaParaOrdenar := Tlista.Create(); //Passada para escopo global
          for x  := 0 to grafo[ProcuraID].TotalDeElementos -1 do
          begin
            ListaParaOrdenar.PoemNaFila(PGrafo.Vertice.nID );
            //Para ordenar os vértices em ordem de preferência
            //será usado o seguinte metodo: a preferência é
            //obtida de acordo com os caminhos que o fantasma
            //encontrou
            Pgrafo.Vertice.Distancia := ceil(10000 /
(Pgrafo.Vertice.Preferencia +1));
            PGrafo := PGrafo.Proximo ;
          end;
         for x  := 0 to grafo[ProcuraID].TotalDeElementos -1 do
         begin
           //Para esvaziar a pilha recurssiva mais rapidamente
           if bpAchouDestino then break;
             //Testar os vértices na ordem de menor custo
             if ListaParaOrdenar.TotalDeElementos() > 0 then
             begin
               nId := ListaParaOrdenar.Proximo();
               PGrafo := grafo[nID];
               ListaParaOrdenar.TiraItem(nID);
             end;
             //Se o vértice ainda não foi testado
             if (PGrafo.Vertice.Passadas = 0) and not(bpAchouDestino) then
            begin
              PGrafo.Vertice.Passadas := 1;
              PGrafo.Vertice.Anterior := grafo[ProcuraID].Vertice ;
              if MODO_DEBUG then
              begin
                if (Pgrafo.Vertice.Debug = self.DebugId) or
```

```
(Pgrafo.Vertice.Debug = 0) then
                begin
                  Pgrafo.vertice.Brush.Color := clred;
                  Pgrafo.Vertice.Debug := self.DebugId;
                end;
              end;
              //Todos os que serão pintados de vermelho, não
              //fazem parte do caminho escolhido e por isso não
              //são preferenciais para as próximas buscas
              Pgrafo.vertice.Preferencia:=Pgrafo.vertice.
Preferencia - 1;
              //Para estatísticas
              TotalDeTestestados := TotalDeTestestados + 1;
              //Ponto de debug :-/
                if TotalDeTestestados = 47 then
                begin
                  TotalDeTestestados := (x*0) + TotalDeTestestados;
                  self.BP_Recurssiva(Pgrafo.Vertice.nID, DestinoId);
                end;
              end;
            end;
           end;
```

Quadro 6 – Código fonte do algoritmo de busca em profundidade

O quadro 7 traz o código fonte da implementação da função melhor vértice que é utilizada pelos algoritmos busca em profundidade adaptado, A* e A* adaptado.

```
function  Tfantasma.MelhorVertice ( ListaAbertos : TLista ) :Integer;
var
  melhor          : Integer; // Recebe o Id do melhor item
  MelhorDistancia : LongWord; // Recebe a melhor distância
  Item            : Integer; // Armazena o item analizado
  ItensTestadosos : integer; // Número de itens que foram
  //testados, para evitar travamentos quando a função próximo
  //retorna 0
begin
  //Posiciona a lista no começo
  ListaAbertos.VaiProInicio ();
  MelhorDistancia := 4294967294; //Distância longa qualquer
  Melhor := 0;  //Irá conter o melhor vértice
  ItensTestadosos := 0; //Número de vértices na lista
  //Passa por todos elementos da lista
  while not ListaAbertos.ChegouNoFim do
  begin
    Item := ListaAbertos.Proximo ();
    //Guarda o de menor distância
    if (grafo[Item].Vertice.Distancia < MelhorDistancia) and
(ItensTestadosos < ListaAbertos.TotalDeElementos()) then
    begin
      Melhor:=Item;
       MelhorDistancia := grafo[Item].Vertice.Distancia;
    end;
    ItensTestadosos := ItensTestadosos +1;
  end;
    result := Melhor; //Retorna o de menor distância
end;
```

Quadro 7 – Código fonte da função melhor vértice

O quadro 8 traz o código fonte da implementação do algoritmo de busca em profundidade adaptado.

```
procedure TFantasma.BP_Recurssiva(ProcuraID :integer; DestinoId: Integer);
var
  x , nID: Integer;
  PGrafo : TListaDeAdjacentes;
  ListaParaOrdenar : TLista;
Begin
  //Indica que a pesquisa já passou por esse vértice
  grafo[ProcuraID].Vertice.Passadas := 1;
  //Agora verifica todos os adjacentes
  PGrafo := grafo[ProcuraID].Proximo;
  //Testa se o destino não foi encontrado
  if ProcuraId = DestinoId then
    bpAchouDestino := true;
    //Para aumentar a performance, quando encontra o destino, nada mais
deve ser feito
    if not bpAchouDestino then
    begin
      //Adaptação que mescla o algoritmo de busca em profundidade com o A*
      //Ordena os vértices adjacentes a ser testados por ordem de menor
custo...
       //Coloca todos os adjacentes não testados em uma lista e calcula a
distância
       ListaParaOrdenar := Tlista.Create(); //Passada para escopo global
       for x  := 0 to grafo[ProcuraID].TotalDeElementos -1 do
       begin
         ListaParaOrdenar.PoemNaFila(PGrafo.Vertice.nID );
         //Estima a distância através da fórmula do A*
         //adicionado um custo experimental do local.
         //O custoLocal do vértice aumenta cada vez que um
         //fantasma passa por ele, e é zerado pelo personagem
         //“come-come”, Isso desistimula os “fantasmas” a andar
         // em círculos, e os estimula a seguir o “come-come”
         Pgrafo.Vertice.Distancia :=
Pgrafo.Vertice.EstimaDistancia(grafo[destinoId].Vertice.X
,grafo[destinoId].Vertice.Y) + pgrafo.Vertice.CustoLocal;
         PGrafo := PGrafo.Proximo ;
       end;
       for x  := 0 to grafo[ProcuraID].TotalDeElementos -1 do
       begin
         //Para esvaziar a pilha recurssiva mais rapidamente
        if bpAchouDestino then
          break;
        //Para testar os vértices na ordem de menor custo
        if ListaParaOrdenar.TotalDeElementos() > 0 then
        begin
          nId := self.MelhorVertice(ListaParaOrdenar);
          PGrafo := grafo[nID];
          ListaParaOrdenar.TiraItem(nID);
        end;
        //Se o vértice ainda não foi testado
        if (PGrafo.Vertice.Passadas = 0) and not(bpAchouDestino) then
        begin
          PGrafo.Vertice.Passadas := 1;
          PGrafo.Vertice.Anterior := grafo[ProcuraID].Vertice ;
          if MODO_DEBUG then
          begin
```

```
            if (Pgrafo.Vertice.Debug = self.DebugId) or
(Pgrafo.Vertice.Debug = 0) then
            begin
              Pgrafo.vertice.Brush.Color := clred;
              Prafo.Vertice.Debug := self.DebugId;
            end;
          end;
          //Para estatísticas
          TotalDeTestestados := TotalDeTestestados + 1;
          //Ponto de debug... :-/
          if TotalDeTestestados = 47 then
            TotalDeTestestados := (x*0) + TotalDeTestestados;
            self.BP_Recurssiva(Pgrafo.Vertice.nID, DestinoId);
          end;
       end;
    end;
  end;
```

Quadro 8 – Código fonte do algoritmo de busca em profundidade adaptado

O quadro 9 traz o código fonte da implementação do algoritmo A*.

```
procedure TFantasma.A_ESTRELA(DestinoX :Integer; DestinoY :Integer);
var
  Abertos, Fechados : TLista; //Lista de vértices
  Achou :Boolean; //Pára após encontrar o caminho
  NovaDistancia :Integer;
  DistanciaDoMelhor :Integer;
  DestinoId, nId : Integer;
  TotalDeAdjacentes, X : Integer;
  MelhorV : Integer;
  PGrafo : TListaDeAdjacentes;
  Plab : TLabirinto;
  nDirecao : Integer;
  Testados : integer;
Begin
  //Inicializa as variáveis
  Achou := False;
  Abertos := TLista.Create();
  Fechados := Tlista.Create();
  Testados := 0;
  //Inicializa os vértices para a busca
  for X:=0 to NumeroDeVertices-1  do
  begin
      //Marca todos os vértices como fora das listas
      grafo[X].Vertice.nEstado:=NENHUM;
      //Marca todos os vértices como estando muito longe
      grafo[X].Vertice.Distancia := 32000;
      if MODO_DEBUG then
        if (grafo[x].vertice.Debug = self.DebugId) or
(grafo[x].vertice.Debug = 0) then
        begin
          grafo[x].Vertice.Brush.Color:=clBlack;
          grafo[x].Vertice.Debug := 0;
        end;
      end;
      //Descobre o ID do destino
      DestinoId := lab[DestinoX,DestinoY].nID ;
      //Coloca o vértice raiz na lista de Abertos
      Abertos.PoemNaFila( lab[self.X,self.Y].nID );
      lab[self.X,self.Y].nEstado := ABERTO; //Feito assim para otimizar...
```

```
      //Define a distância do vértice raiz como zero;
      grafo[lab[self.X,self.Y].nID].Vertice.Distancia := 0;
      while (Abertos.TotalDeElementos()>0) and (not Achou) do
      begin
        //Pega o melhor vértice a ser testado
        MelhorV := self.MelhorVertice(Abertos);
        //Testa se ele não é o vértice de destino
        if MelhorV = DestinoId then
        begin
          Achou := true;  //Indica que o melhor caminho já foi encontrado
        end;
        //Analisa os vértices adjacentes
        TotalDeAdjacentes := grafo[MelhorV].TotalDeElementos ;
        //Recebe o primeiro vértice adjacente do escolhido
        PGrafo := Grafo[MelhorV].Proximo ;
        //Pega a distância do vértice escolhido até o destino
        DistanciaDoMelhor :=
Grafo[MelhorV].Vertice.EstimaDistancia(DestinoX,DestinoY) ;
        //Para todos os adjacentes...
        for X := 1 to TotalDeAdjacentes do
        begin
          if MODO_DEBUG then
          begin
            if (Pgrafo.Vertice.Debug = self.DebugId) or
(Pgrafo.Vertice.Debug = 0) then
            begin
              Pgrafo.vertice.Brush.Color := clred;
              Pgrafo.Vertice.Debug:=self.DebugId;
            end;
          end;
          //Para estatísticas
          Testados := Testados + 1;
          //Calcula a distância do vértice adjacente (Se o vértice contem
um outro fantasma, o custo adicional é 2)
          //Isso evita que os fantasmas fiquem um seguindo o outro
          NovaDistancia := DistanciaDoMelhor +
Pgrafo.Vertice.EstimaDistancia(DestinoX, DestinoY) + Pgrafo.Vertice.nContem
;
          if (Pgrafo.Vertice.nEstado = NENHUM) and (NovaDistancia <
Pgrafo.Vertice.Distancia) then
          begin
            //Marca o vértice anterior
            pgrafo.Vertice.Anterior:= grafo[MelhorV].Vertice ;
            //Coloca a distância calculada
            PGrafo.Vertice.Distancia := NovaDistancia;
            //Se o vértice está na lista de fechados tira
            if PGrafo.Vertice.nEstado = FECHADO then
            begin
              Fechados.TiraItem( Pgrafo.vertice.nID );
              PGrafo.Vertice.nEstado := NENHUM;
            end;
            //Se ainda não estiver na lista de abertos coloca
            if PGrafo.Vertice.nEstado <> ABERTO then
            begin
Abertos.PoemNaFila(Pgrafo.vertice.nID);
              PGrafo.Vertice.nEstado := ABERTO;
            End;
          End;
          //Avança para o próximo vértice adjacente
          PGrafo := Pgrafo.Proximo ;
        end;
```

```
        Abertos.TiraItem(MelhorV);
        //Marca o vértice analisado como fechado
        Fechados.PoemNaFila(MelhorV);
        grafo[MelhorV].Vertice.nEstado := FECHADO;
      end;
      //Total de testes realizados
      TotalArestasAE := TotalArestasAE + Testados;
      str(Testados,S);
      frmLabirinto.lblVertices.Caption:=S;
      //Terminado esse processo, o melhor caminho é o caminho inverso
(Vertice.Anterior) até a raiz
      if (ACHOU)  then
      begin
        //Descobre o Id do destino
        PLab := lab[DestinoX,DestinoY];
        nId := lab[self.X, self.Y].nID;
        //Encontra o melhor caminho para o fantasma
        repeat
          PLab := Plab.Anterior;
          //Pinta de Amarelo o caminho mais curto descoberto
          if MODO_DEBUG then
          begin
            PLab.Brush.Color := clyellow;
            Plab.Debug := self.DebugId ;
          end;
          //Descobre a direção a ser tomada pelo fantasma
          nDirecao := plab.nID;
        until (Plab.Anterior.nID = nId) or (Plab.Distancia = 0);
          pGrafo := grafo[nDirecao];
          //Descobre a direção a tomar
          while pGrafo.Vertice.nID <> nId do
          begin
            Pgrafo := PGrafo.Proximo;
          end;
          self.Direcao:=pGrafo.Direcao  ;
        end;
        //Para no caso de não achar o caminho
        self.Direcao:=self.Direcao;
      end;
```

Quadro 9 – Código fonte do algoritmo A*

O quadro 10 traz o código fonte da implementação do algoritmo A* adaptado

```
procedure TFantasma.SuperBusca(DestinoX: Integer; DestinoY :Integer);
var
  Abertos, Fechados : TLista; //Lista de vértices
  Achou :Boolean; //Pára após encontrar o caminho
  NovaDistancia :Integer;
  DistanciaDoMelhor :Integer;
  DestinoId, nId : Integer;
  TotalDeAdjacentes, X : Integer;
  MelhorV : Integer;
  PGrafo : TListaDeAdjacentes;
  Plab : TLabirinto;
  nDirecao : Integer;
  Testados : integer;
  Niveis : Integer;
begin
  //Inicializa as variáveis
  Achou := False;
```

```
  Abertos := TLista.Create();
  Testados := 0;
  Niveis := 0 ;
  //Inicializa os vértices para a busca
  for X:=0 to NumeroDeVertices -1 do
  begin
    //Marca todos os vértices como fora das listas
    grafo[X].Vertice.nEstado:=NENHUM;
    //Marca todos os vértices como estando muito longe
    grafo[X].Vertice.Distancia := 32000;
    if MODO_DEBUG then
      if (grafo[x].vertice.Debug = self.DebugId) or (grafo[x].vertice.Debug
= 0) then
      begin
         grafo[x].Vertice.Brush.Color:=clBlack;
         grafo[x].Vertice.Debug := 0;
       end;
    end;
    //Descobre o ID do destino
    DestinoId := lab[DestinoX,DestinoY].nID ;
    //Coloca o vértice raiz na lista de abertos
    Abertos.PoemNaFila( lab[self.X,self.Y].nID );
    lab[self.X,self.Y].nEstado := ABERTO; //Para otimizar
    //Define a distância do vértice raiz como zero;
    grafo[lab[self.X,self.Y].nID].Vertice.Distancia := 0;
      while (Abertos.TotalDeElementos()>0) and (not Achou) and (Niveis <
25) do
      begin
        Niveis := Niveis + 1;
        //Pega o melhor vértice a ser testado
        MelhorV := self.MelhorVertice(Abertos);
        //Testa se ele não é o vértice de destino
        if MelhorV = DestinoId then
        begin
          Achou := true;  //Indica que o melhor caminho já foi encontrado
        end;
        //Analisa os vértices adjacentes
        TotalDeAdjacentes := grafo[MelhorV].TotalDeElementos ;
        //Recebe o primeiro vértice adjacente do escolhido
        PGrafo := Grafo[MelhorV].Proximo ;
        //Pega a distância do vértice escolhido até o destino
        DistanciaDoMelhor :=
Grafo[MelhorV].Vertice.EstimaDistancia(DestinoX,DestinoY) ;
        //Para todos os adjacentes
        for X := 1 to TotalDeAdjacentes do begin
          if MODO_DEBUG then
          begin
            if (Pgrafo.Vertice.Debug = self.DebugId) or
(Pgrafo.Vertice.Debug = 0) then
            begin
              Pgrafo.vertice.Brush.Color := clred;
              Pgrafo.Vertice.Debug:=self.DebugId;
            end;
          end;
          //Para estatísticas
          Testados := Testados + 1;
          //Calcula a distância do vértice adjacente
          NovaDistancia := DistanciaDoMelhor +
Pgrafo.Vertice.EstimaDistancia(DestinoX, DestinoY) +
Pgrafo.Vertice.CustoLocal ;
          if (Pgrafo.Vertice.nEstado = NENHUM) and (NovaDistancia <
```

```
Pgrafo.Vertice.Distancia) then
          begin
            //Marca o vértice anterior
            pgrafo.Vertice.Anterior:= grafo[MelhorV].Vertice ;
            //Coloca a distância calculada
            PGrafo.Vertice.Distancia := NovaDistancia;
            //Se ainda não estiver na lista de abertos, coloca
            if PGrafo.Vertice.nEstado <> ABERTO THEN
            begin
              Abertos.PoemNaFila(Pgrafo.vertice.nID);
              PGrafo.Vertice.nEstado := ABERTO;
            end;
          end;
          //Avança para o próximo vértice adjacente
          PGrafo := Pgrafo.Proximo ;
        end;
        //Tira o vértice analisado da lista de abertos
        if grafo[MelhorV].Vertice.nEstado = ABERTO then
        begin
          Abertos.TiraItem(MelhorV);
        end;
        //Marca o vértice analisado como fechado
        grafo[MelhorV].Vertice.nEstado := FECHADO;
      end;
      //Total de testes realizados
      TotalArestasAE := TotalArestasAE + Testados;
      str(Testados,S);
      frmLabirinto.lblVertices.Caption:=S;
      //Agora, depois de testar apenas os adjacentes do vértice origem
      //descobre a direção a seguir, pegando o melhor vértice
      //dentre os abertos. Ele é o próximo passo do fantasma.
      MelhorV := self.MelhorVertice(Abertos);
      //Descobre o Id do destino
      PLab := grafo[MelhorV].Vertice ;
      nId := lab[self.X, self.Y].nID;
      //Encontra o melhor caminho para o fantasma.
      repeat
        PLab := Plab.Anterior;
        //Pinta de amarelo o caminho mais curto descoberto.
        if MODO_DEBUG then
        begin
          PLab.Brush.Color := clyellow;
          PLab.Debug:=self.DebugId;
        end;
        //Descobre a direção a ser tomada pelo fantasma.
        nDirecao := plab.nID;
      until (Plab.Anterior.nID = nId) or (Plab.Distancia = 0);
        pGrafo := grafo[nDirecao];
        //Descobre a direção a seguir.
        while pGrafo.Vertice.nID <> nId do
        begin
          PGrafo := PGrafo.Proximo;
        end;
        self.Direcao:=Pgrafo.Direcao ;
        self.VerticeOndeEuEstava:=lab[self.X,self.Y];
      end;
```

Quadro 10 – Código fonte do algoritmo A* adaptado