Maurício Archanjo Nunes Coelho

Busca de Caminhos em Espaços 3D

Orientador:
Raul Fonseca Neto

Co-orientador:
Marcelo Bernardes Vieira

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Juiz de Fora

Monografia submetida ao corpo docente do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora como parte integrante dos requisitos necessários para obtenção do grau de bacharel em Ciência da Computação

Prof. Raul Fonseca Neto, D. Sc.
Orientador

Prof. Marcelo Bernardes Vieira, D. Sc.
Co-Orientador

Prof. Custódio Gouvêa Lopes da Motta, M. Sc.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Este trabalho trata sobre o problema de planejamento de caminhos, assunto da área de inteligência Artificial. Apresenta uma abordagem sobre planejamento de caminhos em ambientes 3D complexos: estáticos ou dinâmicos, tendo como foco uma entidade agente dividida em dois componentes básicos: corpo e mente. O Agente inteligente busca o melhor caminho que satisfaz suas restrições. Uma implementação de um agente planejando caminhos e movendo-se num ambiente 3D complexo é realizada para ilustrar os resultados.

1 Introdução

A Inteligência Artificial(I.A.) é uma das ciências mais recentes, abrangendo diversas outras disciplinas. O progresso recente na compreensão da base teórica da inteligência artificial caminha lado a lado com os avanços na capacidade de sistemas reais. Os subcampos da I.A. se tornaram mais integrados, e a I.A. encontrou uma área de concordância com outras disciplinas.

O *software* para inteligência artificial resolve problemas complexos que não são passíveis de computação ou análise direta. Aplicações nessa área incluem robótica, sistemas especialistas, reconhecimento de padrões (de imagem e voz), redes neurais, e prova de teoremas e jogos (PRESSMAN, 2006).

A Inteligência Artificial está em pleno desenvolvimento e possui diversas linhas de pesquisa, e uma área que está em ascensão é a simulação de entidades inteligentes em ambientes 3D.

1.1 Inteligência Artificial e Computação Gráfica

Uma outra área que tem avançado muito, graças a um maior poder de processamento e métodos de modelagem 3D cada vez mais eficientes é a computação gráfica. Atualmente simulando com um realismo impressionante estruturas, pessoas e ambientes.

Com o auxílio de métodos dessas duas áreas conseguimos unir a mente de uma entidade a seu corpo, visualizando seu comportamento. Assim, podemos simular agentes, como são chamados essas entidades em I.A., na forma de personagens animados atuando em ambientes virtuais.

Este trabalho conta com o apoio do projeto de monografia: Gerenciamento e Visualização de Ambientes Virtuais, do formando pela Universidade Federal de Juiz de Fora, Lucas Grassano Lattari, cuja implementação da modelagem do ambiente 3D e do corpo do agente foram vitais para a visualização da busca e possibilitou uma melhor modelagem

da racionalidade do agente.

1.2 Definição do Problema

O problema abordado neste trabalho é de como planejar caminhos para que uma entidade agente se locomova de maneira racional e aparentemente natural num ambiente virtual 3D complexo.

Devem ser levados em consideração quais métodos de busca são mais eficientes e qual tipo de agente é o mais indicado para o ambiente de tarefa no qual se deseja trabalhar. A busca de caminhos e a locomoção do agente devem ser realizadas eficientemente, pois a limitação do hardware é um fator importante a considerar.

1.3 Proposta de Trabalho

Até alguns anos atrás, a Inteligência Artificial era considerada pura e estritamente acadêmica. O desenvolvimento de inteligência nas máquinas vinha sendo motivado, em sua maior parte, pelos anseios da própria comunidade científica.

Nos últimos anos, entretanto, um dos estímulos para o desenvolvimento da Inteligência Artificial no âmbito comercial e empresarial foi a indústria de desenvolvimento de jogos para computador. A avançada tecnologia de computação gráfica em tempo real torna os cenários e personagens dos jogos muito mais realistas, fazendo com que o jogador passe a esperar o mesmo nível de realidade no comportamento dos personagens e de outras entidades controladas pelo computador.

Isto vem motivando o desenvolvimento de diversas áreas da I.A. como a de Agentes Inteligentes, Aprendizado de Máquina e Aquisição de Conhecimento. Sistemas de agentes inteligentes para coordenar as ações dos personagens do jogo controlados pelo computador passaram então a ser necessários. A interação dos personagens com o jogador, e as reações daqueles em virtude das atitudes deste, passaram a ser governadas por um sistema autônomo que, em alguns casos, é capaz de aprender com situações passadas, adaptando-se dinamicamente às novas situações de jogo (DELOURA, 2000).

Porém no meio acadêmico a inteligência artificial aplicada para esse fim passou desapercebida durante algum tempo. Então torna-se necessário uma abordagem de cunho acadêmico para um estudo mais aprofundado e seu desenvolvimento nos moldes científicos.

E é nesse contexto no qual este trabalho se insere.

O objetivo principal deste trabalho é pesquisar, analisar e formalizar os fundamentos e métodos necessários para que um agente inteligente possa planejar caminhos em ambientes virtuais eficientemente, baseado em suas próprias percepções. Para isso são apresentados estudos sobre agentes inteligentes, ambientes de tarefas e métodos de buscas variados, entre outros.

O objetivo secundário é modelar e implementar agentes inteligentes capazes de se locomoverem racionalmente e com certo grau de autonomia em ambientes 3D variados, atingindo um objetivo pré-determinado.

1.4 Organização do Trabalho

O Capítulo 2 apresenta a matemática necessária para o entendimento do trabalho.

Os Agentes Inteligentes, vistos no Capítulo 3, apresentam os vários tipos de agentes possíveis dentro da I.A..

No Capítulo 4 temos a explicação de diversos tipos de buscas, informadas e não-informadas. Seguida da explicação de heurísticas.

Segue-se o Modelo Computacional no Capítulo 5. Aqui é visto como computacionalmente as buscas são feitas, e são epresentadas as buscas que foram implementadas.

No Capítulo 6, os resultados da aplicação podem ser observados, ilustrando o conhecimento aplicado. Diagramas e tabelas são apresentados para ilustrar o funcionamento do programa.

Finalmente, no Capítulo 7, algumas conlusões sobre o trabalho são apresentadas juntamente com as possibilitades de trabalhos futuros.

2 *Noções Matemáticas Introdutórias*

O objetivo deste capítulo é contextualizar esse trabalho dentro da grande área da I.A. abordando seus elementos matemáticos. Para isso vamos apresentar 3 importantes elementos: grafos, sistemas dinâmicos e otimização combinatória, com ênfase no algoritmo de Dijkstra.

2.1 Grafos

O grafo é uma representação gráfica das relações existentes entre elementos de um conjunto. Ele pode ser descrito num espaço euclidiano de n dimensões como sendo um conjunto V de vértices, também conhecidos como nós, ligados por um conjunto E de curvas contínuas (arestas ou arcos). Dependendo da aplicação, as arestas podem ser direcionadas, e são representadas por setas. Então o grafo é um par ordenado representado por $G = (V, E)$. (CORMEN; RIVEST, 2000) O grafo dentro do contexto de otimização combinatória é comumente chamado de rede.

2.2 Sistemas dinâmicos

Um sistema dinâmico é um modelo geral dos sistemas que evoluem segundo uma regra que liga o estado presente aos estados passados. Um sistema dinâmico pode ser representado por um conjunto de equações que especificam o modo como as variáveis se alteram ao longo do tempo, ou seja, definem o fluxo do sistema. O estado do sistema é definido pelo valor das suas n variáveis, podendo ser representado por um ponto no espaço de fase: um espaço multidimensional constituído pelos pontos que correspondem aos estados (MONTEIRO, 2002). Em I.A. o espaço de fase é geralmente representado através de um grafo.

Um sistema dinâmico discreto, é um sistema em que o seu estado só muda durante os instantes $\{t_0, t_1, t_2, \ldots\}$. No intervalo de tempo entre dois desses instantes, o estado permanece constante. O intervalo de tempo entre dois instantes sucessivos t_n e t_{n+1} não precisa ser constante. A equação de evolução: $y_{n+1} = f(y_n)$ permite calcular o estado y_{n+1}, num instante $n+1$, a partir do estado y_n, no instante anterior n (VILLATE, 2006).

2.3 Otimização Combinatória

Problemas de otimização objetivam maximizar ou minimizar uma função definida sobre um domínio. A teoria clássica de otimização trata do caso em que o domínio é infinito. No caso dos problemas de otimização combinatória, o domínio é tipicamente finito. Em geral é fácil listar os seus elementos e testar se um dado elemento pertence a esse domínio. Porém, testar todos os elementos deste domínio na busca pelo melhor mostra-se inviável na prática para a maioria dos problemas (GOLDBARG; LUNA, 2005).

Como exemplos temos o problema da mochila, o problema do caixeiro viajante e o problema da satisfabilidade máxima. Eles possuem várias aplicações práticas: projeto de redes de telecomunicação, o empacotamento de objetos em **containers**, a localização de centros distribuidores, análise de dados, na economia (matrizes de entrada/saída), na física (estados de energia mínima), entre outras.

O caminho de custo mínimo é a seqüência de ligações que se deve seguir do nó inicial até o nó final num grafo, cujo custo total é mínimo. É um dos problemas mais estudados em Otimização Combinatória. Existem quatro tipos de situações no problema do caminho mínimo, porém abordaremos só a primeira:

- caminho mínimo entre origem e destino;
- caminho entre a origem e os demais vértices;
- caminho mínimo entre todos os pares de vértices;
- k-ésimos caminhos mínimos entre um par de vértices: contigenciamento.

Princípio da otimalidade: uma seqüência ótima de decisões tem a propriedade de que quaisquer que sejam o estado e a decisão inicial, as decisões remanescentes constituem uma seqüência ótima de decisões com relação ao estado decorrente da primeira decisão, ou seja, toda subtrajetória da trajetória ótima é ótima com relação a suas extremidades inicial e final (BELLMAN, 2003).

Seja $C_{k,j}$ o custo do arco (k, j), $P_{i,j}$ o caminho mínimo de um nó de origem i até um nó destino j que passa por algum nó k em uma rede, e $M_{i,j}$ o somatório dos custos dos arcos do caminho $P_{i,j}$. Podemos dizer pelo princípio da otimalidade que para cada $k \neq j$ existe um arco (k, j), tal que o caminho $M_{i,j} = M_{i,k} + C_{k,j}$ é o menor possível, para todas as possibilidades do nó k. Então podemos dizer que $M_{i,j} = Min(M_{i,k} + C_{k,j})$, $k \neq j$. Procura-se então minimizar a função objetivo, que neste caso é o somatório dos custos dos arcos entre uma origem e um destino: $\sum_i^j M_{i,k} + C_{k,j}$ para cada k. **O problema está em determinar o nó *k*** de maneira eficiente de modo que o caminho seja mínimo.

Na Figura 1 o caminho mínimo (ótimo) entre a e e é o caminho que passa pelos vértices a, b, c, d, e e pelo princípio da otimalidade qualquer subcaminho entre eles também é ótimo, por exemplo b, c, d ou c, d, e.

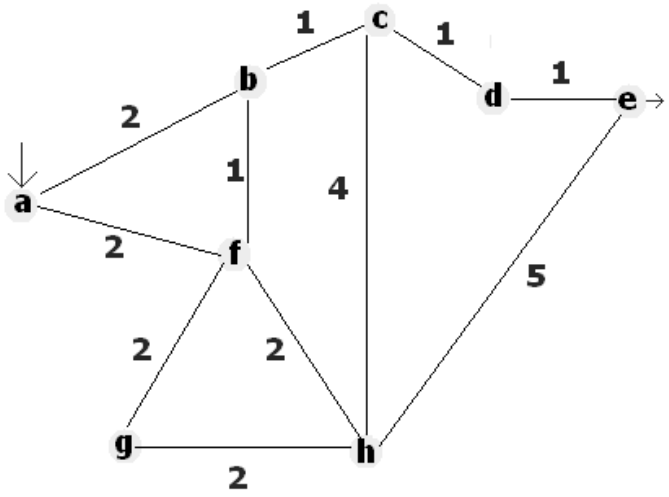

Figura 1: Exemplo de um grafo com custos nas arestas

Neste trabalho, temos $C_{d,e} \in \mathbb{R}^+ \ \forall \ d, e \in V$, e um arco cujo custo é proibitivo possui valor infinito: $C_{f,d} = \propto, \ f, d \in V$.

O problema do caminho mínimo entre dois vértices fixos pode ser modelado como um problema de fluxo compatível a custo mínimo. A quantidade de fluxo que passa no arco (i, j) é representado como $X_{i,j}$. Neste trabalho considera-se a passagem de um fluxo unitário na rede, deste modo o valor da função objetivo retrata o custo do caminho, ou seja, $X_{i,j} \in \{0, 1\}$ então se $X_{i,j} = 0$ o fluxo não passa pelo arco (i, j) e se $X_{i,j} = 1$ o fluxo passa pelo arco. Não há restrições de capacidade no arco quando o fluxo é unitário, simplesmente passa ou não passa, ou seja, o fluxo é sempre compatível.

2.4 Algoritmo de Dijkstra

Dentro da área de otimização combinatória, o método mais usado para encontrar o caminho mínimo entre uma origem pré-fixada e os demais nós da rede, quando não há arcos de custo negativo, é o **Algoritmo de Dijkstra**, ou seja, o algoritmo de Dijkstra identifica, a partir de um nó da rede, qual é o custo mínimo entre esse nó e todos os outros nós da rede. A cada iteração m o algoritmo determina o caminho mínimo de um nó origem i até um nó k qualquer. Esse algoritmo segue o princípio de uma busca ordenada, portanto os custos dos caminhos tem valores crescentes, por esse motivo que não pode haver arcos com custo negativo. Porém isso não chega a ser um grande problema, pois os custos dos arcos são geralmente grandezas físicas mensuráveis.

O algoritmo de Dijkstra considera um grafo $G = (V, E)$, onde os nós petencentes a V são divididos em três conjuntos: os já visitados(conjunto fechados), os candidatos(conjunto abertos) e os não-visitados(conjunto desconhecidos).

Seja $D^m_{i,k}$ a soma dos custos dos arcos para de i se chagar a k passando por um caminho qualquer, e m a m-ésima iteração. Temos que $D^m_{i,k} = Min\{D^{m-1}_{i,p}, D^{m-1}_{i,p} + C_{p,k}\}$. Onde p é um nó fechado na última iteração. O nó cujo $D^m_{i,k}$ foi calculado é colocado no conjunto fechados e seus arcos apontam para os nós que serão incluídos no conjunto abertos. O algoritmo de Dijkstra foi desenvolvido para resolver problemas em rede genéricas.

2.5 Contextualização na I.A.

Na solução de problemas em I.A. por problemas de busca, a determinação do caminho mínimo necessita de uma constante avaliação dos estados dos nós, onde em alguns casos podem ter uma componente de avaliação heurística: a estimativa *h(k)*.

Quando a função objetivo com as restrições citadas acima é incluída no ramo da I.A., a mesma passa a ser chamada de *função de avaliação* e passa a ter a seguinte notação: $f(k) = g(k) + h(k)$ considerando k um nó qualquer da rede. Onde *g(k)* corresponde ao custo exato do caminho desde o nó inicial i até o nó k, se *g(k)* for parte do caminho mínimo teremos que $g(k) = M_{i,k}$. Ora, o custo mínimo do nó k até o nó final j, $C_{k,j}$, não é conhecido na maioria dos problemas de I.A. Assim, o *h(k)* é uma **estimativa** do $C_{k,j}$ visando o caminho mínimo do nó k até o objetivo. Se *h(k)* é uma estimativa adequada ao problema, então *f(k)* é o custo estimado da solução de custo mais baixo passando por k.

Temos então um sub-problema do algoritmo de Dijkstra, este sub-problema calcula o

caminho mínimo entre um nó origem e um nó destino pré-determinados com o auxílio de uma estimativa *h(k)*, ou seja, a inserção desta estimativa no cálculo do caminho mínimo restringe o problema de se calcular o caminho mínimo de um nó origem para os demias nós da rede, o Algoritmo de Dijkstra, em um problema de cálculo de caminho mínimo de um nó origem até um nó destino pré-estabelecidos.

Segundo a equação de evolução $y_{n+1} = f(y_n)$ (VILLATE, 2006) temos que a função $f(k) = g(k) + h(k)$ corresponde a um sistema no qual $f(y_n) = g(k) + C_{k,k+1}$, então $g(k+1) = g(k) + C_{k,j}$, então temos que $f(k+1) = g(k+1) + h(k+1) \Rightarrow f(k+1) = g(k) + C_{k,j} + h(k+1)$, formando portanto um sistema dinâmico.

Se não for levado em conta $h(k)$, ou seja, $h(k) = 0$, temos que $f(k) = g(k)$. Pegando-se o menor valor de $f(n)$ para n distintos, ou seja, minimizando $f(k)$ estaremos levando em conta só os custos do caminho até um k qualquer, ou seja, o mesmo princípio do algoritmo de Dijkstra.

Por outro lado, se não for levado em conta $g(k)$, ou seja, $g(k) = 0$, temos que $f(k) = h(k)$. Pegando-se o menor valor de $f(n)$ para n distintos, ou seja, minimizando $f(k)$ estaremos levando em conta só as estimativas do caminho de k até o nó final j. No capítulo 4, será visto como estimar essa componente heurística $h(k)$ de modo correto para que o caminho seja mínimo.

3 Agentes Inteligentes

Neste capítulo são discutidos os diferentes tipos de agentes e os ambientes onde eles atuam.

"Um agente é tudo o que pode ser considerado capaz de perceber seu ambiente por meio de sensores e de agir sobre esse ambiente por intermédio de atuadores" (Fig. 2). (RUSSEL; NORVING, 2004)

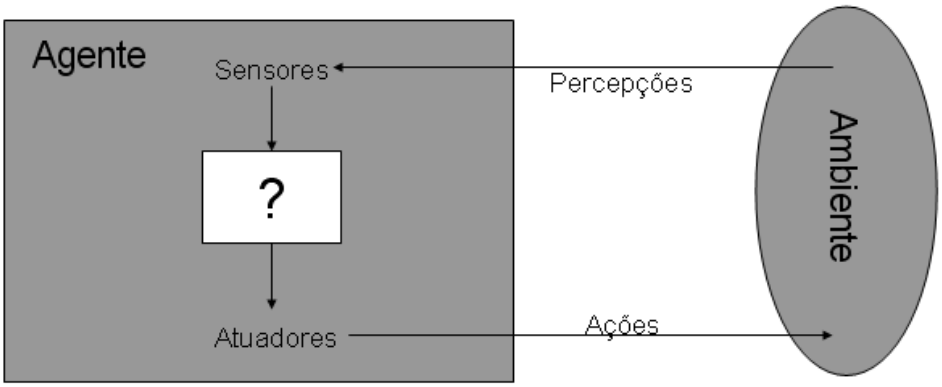

Figura 2: Agentes interagem com ambientes por meio de sensores e atuadores

Como exemplo temos um agente simulando um ser humano. Seus sensores seriam os cinco sentidos: visão, audição, paladar, olfato e tato. E como atuadores teríamos suas articulações, principalmente as pernas e os braços atuando no ambiente.

Quando um agente visa maximizar suas chances de sucesso é dito que tal agente possui algum grau de racionalidade.

A definição de racionalidade depende de quatro fatores: a medida de desempenho que define o critério de sucesso, o conhecimento anterior que o agente tem do ambiente, as ações que o agente pode executar e a sequência de percepções do agente até o momento.

Com base nos quatro fatores anteriores tem-se a seguinte definição de um agente racional segundo (RUSSEL; NORVING, 2004): "Para cada sequência de percepções possível, um agente racional deve selecionar uma ação que se espera venha a maximizar sua medida

de desempenho, dada a evidência fornecida pela sequência de percepções e por qualquer conhecimento interno do agente."

Nos modelos de agentes inteligentes parte-se do princípio de que a inteligência, de alguma forma, já deve estar presente nos elementos autônomos.

"Agência representa o grau de autonomia e autoridade incorporadas ao agente. À medida que o ambiente no qual o agente atua torna-se cada vez mais imprevisível, inovador, torna-se necessário incorporar mais agência ao agente. Esta incorporação de agência é feita através da construção e atualização de modelos mentais que eventualmente são manipulados internamente pelo agente. Ao receber estímulos do ambiente onde está inserido, ou através de uma decisão interna, o agente atualiza o estado de seus modelos mentais, que é o processo de cognição." (FERNANDES, 2000)

O grau de agência dado ao agente deve ser comparada ao nível de dificuldade da tarefa pretendida. Se for preciso modelar a mente de um agente com capacidades próximas a de uma mente humana, por exemplo, necessita-se criar um modelo muito complexo. Por outro lado, se o agente tem apenas que consultar algumas bases de dados na **Web**, com intenção de pesquisar preços, então o seu modelo mental e suas capacidades cognitivas podem ser bastante rudimentares.

Depois do estudo feito sobre os agentes inteligentes e sua racionalidade, é preciso definir um meio para medir seu desempenho.

O critério para se medir o sucesso do comportamento do agente é conhecido como medida de desempenho. Quando um agente é inserido em um ambiente, ele gera uma sequência de ações, de acordo com as percepções que recebe. Essa sequência de ações faz o ambiente passar por uma sequência de estados. Se ela é desejável, o agente funcionou bem.

Porém, não existe uma medida fixa de desempenho que abrange todos os agentes. O projetista é o responsável por criar essas medidas de acordo com o resultado desejado no ambiente.

Existem diferentes tipos de agentes. Porém, antes de explicá-los torna-se necessário especificar a natureza dos ambientes onde os agentes atuam.

3.1 Ambiente de Tarefa

Ao projetar um agente, é preciso especificar o ambiente de tarefa tão completo quanto as necessidades do agente exigem. *Um ambiente de tarefa é essencialmente o problema para quais os agentes racionais são as soluções* (RUSSEL; NORVING, 2004).

O termo ambiente de tarefa é definido como um agrupamento da medida de desempenho, do ambiente, dos sensores e dos atuadores, e apresenta as seguintes propriedades:

- Completamente observável *versus* parcialmente observável: Se o agente tem acesso total ao ambiente a todo instante, o ambiente é completamente observável, caso contrário é parcialmente observável;
- Determinístico *versus* estocástico: é determinístico se a ação executada pelo agente mais o estado atual do agente são suficientes para determinar o próximo estado do ambiente, caso contrário é estocástico;
- Episódico *versus* sequencial: a experiência do agente é dividida em episódios atômicos. Se para todos os episódios o episódio seguinte não depender dos anteriores, é episódico, caso contrário é sequencial;
- Estático *versus* dinâmico: Se o ambiente puder se alterar enquanto o agente está deliberando, o ambiente é dinâmico, caso contrário é estático;
- Discreto *versus* contínuo: refere-se a como o tempo é tratado.
- Agente único *versus* multiagente: Se um agente reconhece outra entidade como agente e coopera, compete ou estabelece algum tipo de comunicação com ele o ambiente é considerado multiagente, caso contrário pode ser considerado como agente único.

Os agentes devem, preferencialmente, possuir sensores que captem somente as informações do ambiente de tarefa que visem maximizar seu desempenho. Um agente que está no vácuo do espaço, não necessita do sensor 'audição', por exemplo. Ao analisar as informações, defini-se que ações serão executadas pelos atuadores, porém, os atuadores precisam ter condições de realizar a ação estipulada. Por exemplo, não adianta escolher a ação 'andar' se o agente não possuir nenhum dispositivo para sua locomoção.

Depois da definição dos tipos de ambientes de tarefas torna-se possível a descrição dos tipos básicos de agentes.

3.2 Tipos de Agentes

Existem quatro tipos básicos de programas de agentes: reativos simples, reativos baseados em modelo, baseados em objetivos e baseados na utilidade. Um caso especial são os agentes com capacidade de aprendizagem.

3.2.1 Agentes Reativos Simples

São os mais simples, selecionam ações com base na percepção atual, ignorando o restante histórico de percepções. Devido a sua simplicidade, caracteriza-se por ter uma inteligência muito limitada. Não é possível utilizá-lo eficientemente em ambientes complexos (Fig. 3).

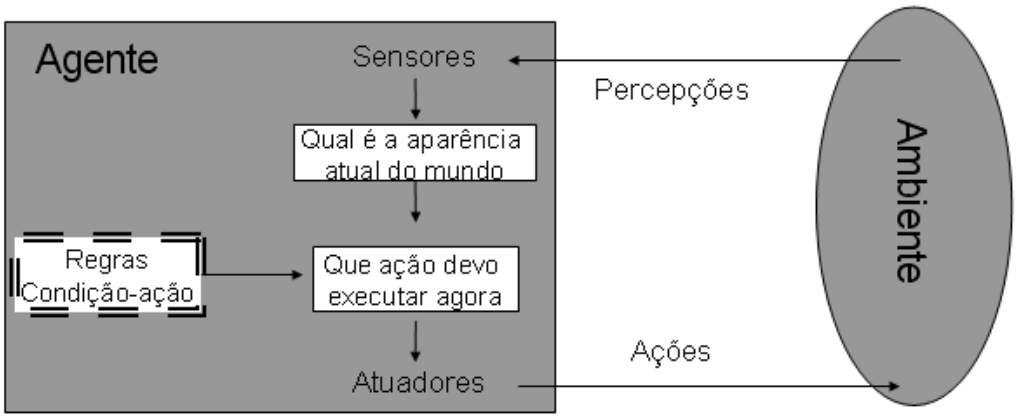

Figura 3: Diagrama esquemático de um agente reativo simples

3.2.2 Agentes Reativos Baseados em Modelos

O Agente mantém algum tipo de estado interno dependente do histórico de percepções. Como exemplo temos um homem em uma caverna, a cada três encruzilhadas, uma ele vira para a esquerda e duas para a direita, nessa ordem, ele precisa guardar seu estado atual: onde virou da última vez; e usar um modelo interno para escolher a ação.

Ele controla o estado atual do mundo usando algum tipo de modelo interno. Em seguida, ele escolhe uma ação da mesma maneira que o Agente Reativo Simples. Ele é eficiente em ambientes parcialmente observáveis, pois através de seu modelo, controla uma parte do mundo que não pode ser vista no momento (Fig. 4).

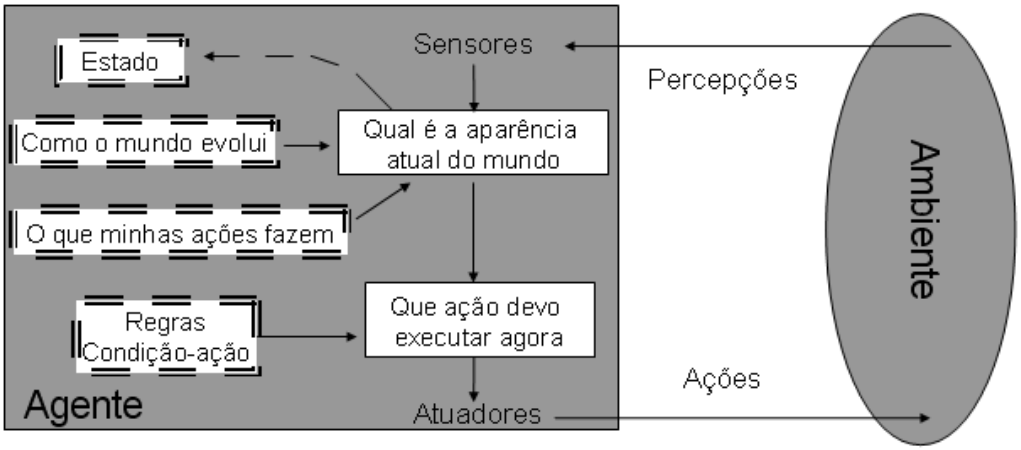

Figura 4: Diagrama esquemático de um agente reativo baseado em modelo

3.2.3 Agentes Baseados em Objetivos

Conhecer o estado atual do ambiente nem sempre é suficiente para se decidir o que fazer, por exemplo, um homem dentro de uma caverna, num entrocamento ele precisaria saber qual caminho seguir se ele quiser sair da caverna.

O agente combina informações que descrevem situações desejáveis com informações sobre os resultados de ações possíveis, a fim de escolher ações que alcancem os objetivos. para isso o agente deve considerar longas sequências de ações até encontrar um meio de atingir o objetivo, sendo necessário algum tipo de busca (Fig. 5).

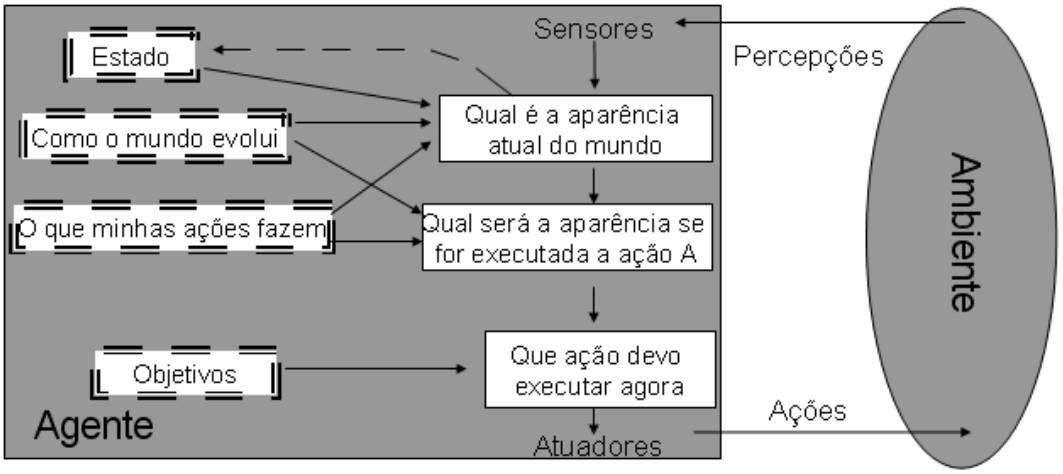

Figura 5: Diagrama esquemático de um agente baseado em objetivo

3.2.4 Agentes Baseados na Utilidade

Os objetivos permitem apenas um distinção entre estado esperado e estado qualquer. No Agente Baseado em utilidades existe uma função de utilidade que mapeia um estado em um número real e descreve o grau de felicidade associado.

Como exemplo podemos continuar com o homem na caverna, porém agora ele está caçando tesouros e num dos caminhos existem animais selvagens e logo atrás um baú, a função de utilidade irá fornecer um meio pelo qual a probabilidade de sucesso pode ser ponderada em relação à importância dos objetivos, ou seja, no exemplo dado, seguir ou não o caminho depende do tamanho do tesouro e do grau de periculosidade dos animais (Fig. 6).

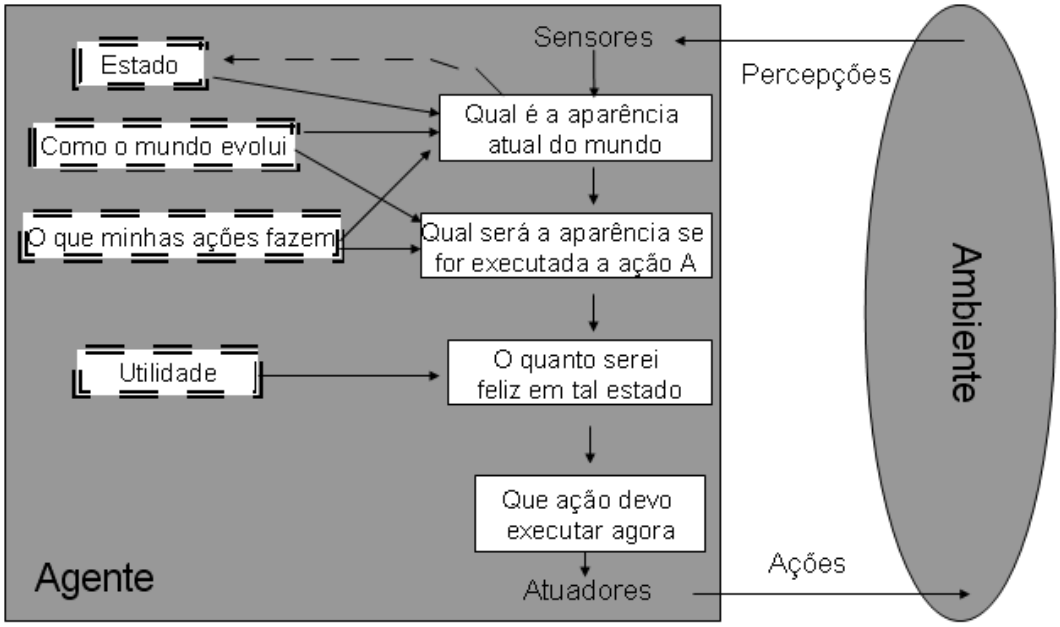

Figura 6: Diagrama esquemático de um agente baseado na utilidade

3.2.5 Agentes com Aprendizagem

O aprendizado permite ao agente operar em ambientes inicialmente desconhecidos e se tornar mais competente do que se tivesse apenas seu conhecimento inicial. Qualquer agente, inclusive os já citados, pode tirar vantagem da aprendizagem. Existe uma grande variedade de métodos de aprendizado.

O aprendizado em agentes inteligentes pode ser resumido como um processo de modificação de cada componente do agente, a fim de tornar mais preciso as informações de

realimentação disponíveis, melhorando assim o desempenho global do agente. Não será discutido aqui como se dá esse aprendizado pois foge ao escopo deste trabalho (Fig. 7).

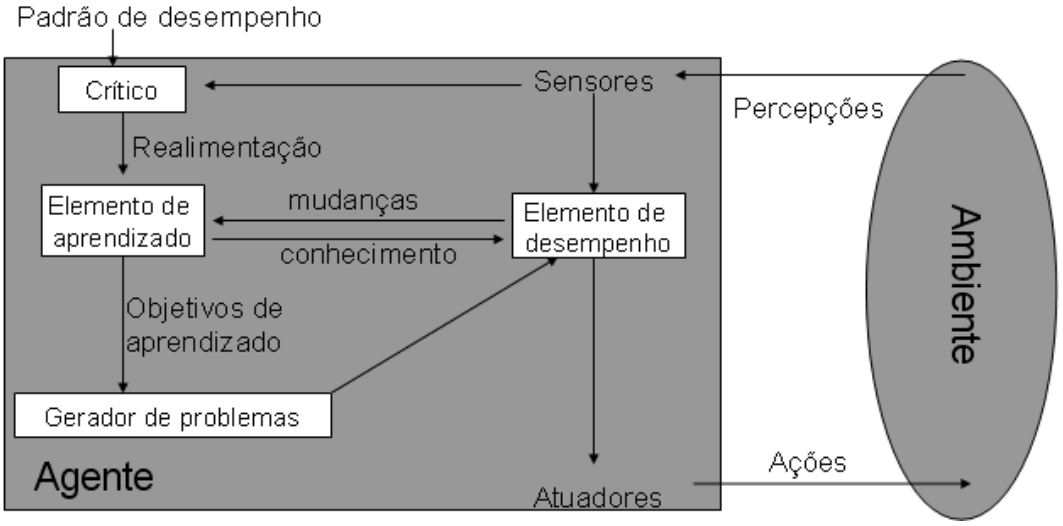

Figura 7: Um modelo geral de agentes com aprendizagem

4 Resolução de Problemas por meio da Busca

As buscas descritas aqui referem-se ao tipo de agente baseado em objetivo. Os agentes decidem o que fazer encontrando uma sequência de ações que levam a estados desejáveis quando nenhuma ação isolada é capaz de fazê-lo. Porém, se o exemplo do agente no labirinto levasse em consideração muitos detalhes nas ações, como mover perna esquerda 10 cm para frente, ele nunca chegaria ao objetivo, pois haveria muitos passos a considerar. Portanto deve-se formular o problema de maneira sensata.

A formulação de problemas é um processo para decidir quais ações e estados devem ser considerados. Depois segue-se a busca: um processo de procurar uma sequência de ações e estados que o leve a seu objetivo.

Um algoritmo de busca recebe um problema como entrada, o ambiente do problema é representado por um espaço de estados. O algoritmo retorna uma solução sob uma forma de sequência de ações, com isso temos um projeto que consiste em formular, buscar e executar. Um problema pode ser definido formalmente em quatro componentes (RUSSEL; NORVING, 2004):

- O estado inicial em que o agente começa;
- Uma descrição das ações possíveis que estão disponíveis para o agente;
- O teste de objetivo, que determina se um estado é o objetivo;
- E uma função de custo de caminho, que atribui um valor numérico a cada caminho refletindo sua própria medida de desempenho.

Conceitos básicos para se entender um problema de busca: ao invés de usarmos a notação $C_{x,y}$ para representar o custo do caminho, usaremos *c(x,a,y)*, porque agora levamos em conta qual ação a foi executada para ir do estado x ao y. Uma solução para um problema de busca é um caminho desde o nó inicial até o final, e uma solução ótima

tem o menor custo de caminho entre todas a outras. O espaço de estados é divididos em três conjuntos: os já visitados: **conjunto Fechados**, os candidatos: **conjunto Abertos**) e os não-visitados: **conjunto Desconhecidos**. E o custo estimado do caminho mais econômico de um estado x até um estado objetivo y é conhecido como componente heurística. São apresentadas agora as definições de alguns termos que medem o desempenho da busca.

4.1 Medição de Desempenho da Busca

Para medir o desempenho durante uma busca temos quatro fatores a considerar:

- Completeza: O algoritmo encontra uma solução se a mesma existir;
- Otimalidade: A solução retornada é ótima;
- Complexidade de tempo: quanto tempo leva para retornar uma solução;
- Complexidade de espaço: quanta memória é necessária pra efetuar a busca.

A Complexidade depende do fator de ramificação no espaço de estados, representado por b, e pela profundidade da solução, representado por d.

A determinação de um caminho e ótimo entre dois nodos em uma rede é um problema fundamental que recebeu atenção considerável de várias comunidades de pesquisa nos últimos quarenta anos, pois suas complexidades de tempo e espaço tornam a resolução de certos problemas impraticável. (WINK; NIESSEN; VIERGEVER, 2000).

Existem dois grandes conjuntos de tipos de buscas: as buscas sem informações, e as buscas com informação, tais buscas possuem desempenhos diferentes . Agora são descritas cada uma delas.

4.2 Buscas sem Informação

São buscas sem informações sobre o espaço de estados, onde eles encontram a solução para problemas gerando sistematicamente novos estados e testando-os por comparação com o objetivo. Agora são descritos algumas dessas buscas.

4.2.1 Busca em Largura

A busca em largura (Fig. 8), seleciona para a expansão o nó menos recente não-expandido na árvore de busca. Ela é completa, ótima para custo unitário e tem complexidade de tempo e espaço iguais a $O(b^d)$. Onde d é a profundidade mínima do caminho. A complexidade de espaço a torna impraticável na maioria dos casos.

A busca de custo uniforme é semelhante à busca em largura, mas expande o nó com caminho de custo mais baixo, *g(n)*. Ela é completa e ótima se o custo de cada passo excede algum limite positivo *E*.

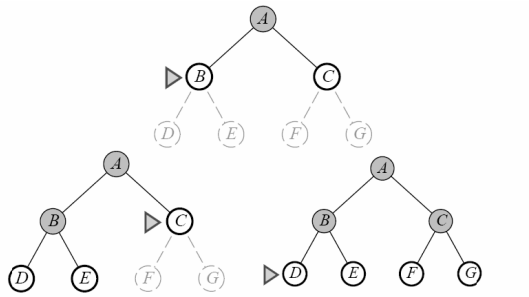

Figura 8: Nós gerados durante uma busca em largura

4.2.2 Busca em Profundidade

A busca em profundidade (Fig. 9), seleciona para a expansão o nó não-expandido mais recente na árvore de busca. Ela não é completa nem ótima, e tem complexidade de tempo igual a $O(b^m)$ e complexidade de espaço igual a $O(bm)$, onde m é a profundidade máxima de qualquer caminho no espaço de estados. A busca em profundidade limitada impõe um limite fixo de profundidade em uma busca em profundidade.

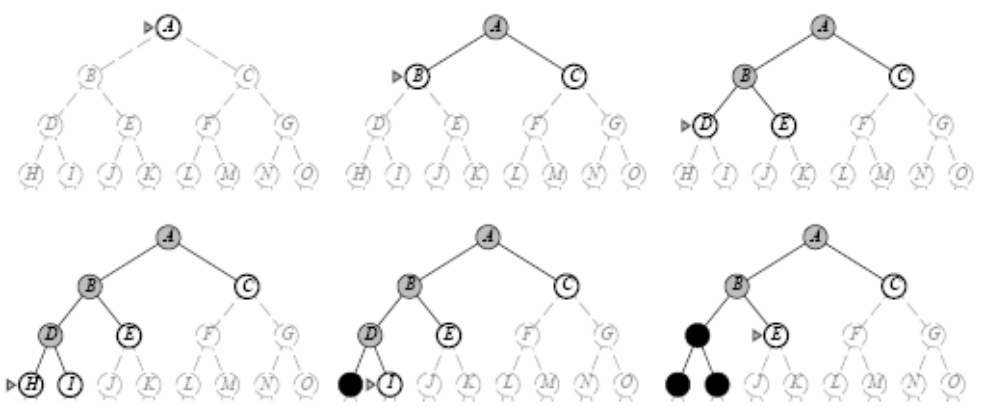

Figura 9: Nós gerados durante uma busca em profundidade

4.2.3 Busca por Aprofundamento Interativo

A busca por aprofundamento interativo (Fig. 10), chama a busca em profundidade com limites crescentes até encontrar um objetivo. Ela é completa, ótima para passos de custo unitário e tem complexidade de tempo igual a $O(b^{d+1})$ e complexidade de espaço igual a $O(bd)$.

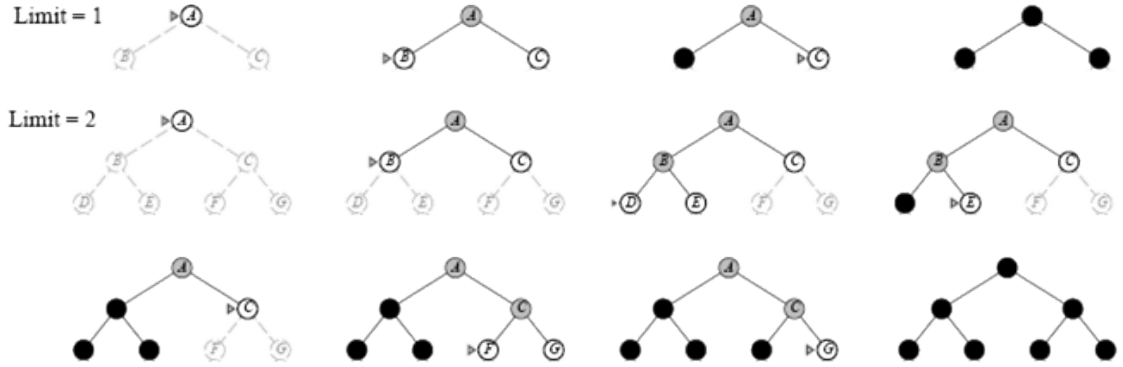

Figura 10: Nós gerados durante uma busca por aprofundamento iterativo

4.2.4 Busca Bidirecional

Realiza simultaneamente duas buscas: a primeira é direta da raiz até o objetivo, a segunda é inversa do objetivo para a raiz (Fig. 11).

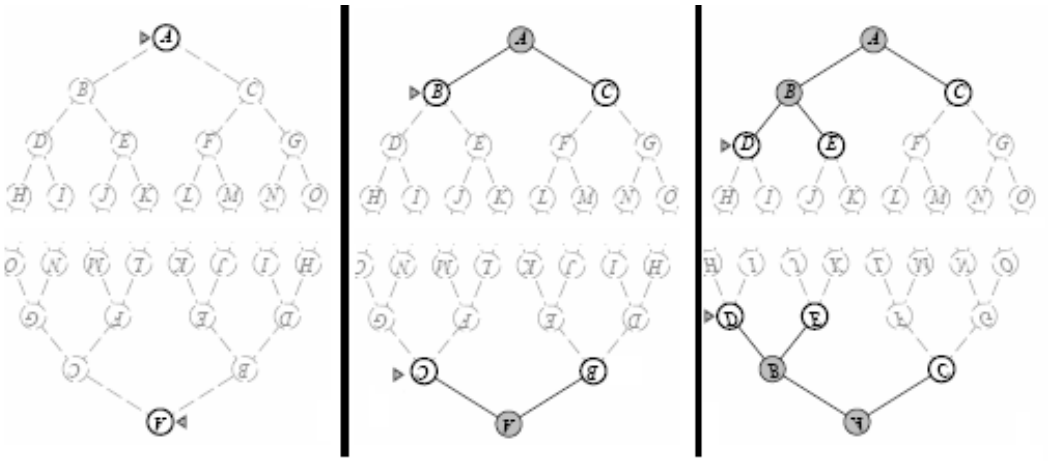

Figura 11: Nós gerados numa busca bidirecional tendo como base a busca em profundidade

A busca bidirecional encontra solução quando as duas frentes se interceptam. Ela pode reduzir enormemente a complexidade de tempo, mas nem sempre é aplicável e pode exigir muito espaço, pois expande e armazena mais nós que uma busca direta.

4.2.5 Busca em Grafos

Quando o espaço de estados é um grafo e se tem a representação completa do mesmo, efetuamos a busca com utilização de uma lista de adjacência e um instrumento de marcação que aponta a aresta ou vértice já visitado. O algoritmo busca em grafo elimina a possibilidade de visitas duplicadas.

Em alguns casos a quantidade de estados repetidos pode fazer um problema se tornar insolúvel, por exemplo, em uma malha retangular, cada estado tem quatro sucessores, e assim a árvore de busca que inclui estados repetidos tem 4^d folhas; porém, existem apenas cerca de $2d^2$ estados distintos dentro de d passos de qualquer estado dado. Com $d = 20$ temos aproximadamente um trilhão de nós no primeiro caso e no segundo apenas 800 estados distintos.

4.3 Buscas com Informação

É utilizado o conhecimento específico do problema, além da definição do próprio problema, através de uma função de avaliação que analisa os méritos do estado em questão.

4.3.1 Busca *Best-First*

Essa busca tenta expandir o nó mais próximo ao objetivo. A mesma escolhe o nó a ser expandido observando os valores da componente heurística *h(n)*, através da seguinte função de avaliação: $f(n) = h(n)$. Porém, ao escolhermos o nó a expandir com base em $g(n)$ ao invés de $h(n)$ temos $f(n) = g(n)$, temos portanto o esquema de busca do próprio algoritmo de Dijkstra.

O uso da função de avaliação permite o estabelecimento de uma ordem de preferência nos estados abertos. Em problemas de minimização o estado de menor valor entre todos é o melhor, já em problemas de maximização o estado de maior valor é o melhor.

Essa abordagem expande nós desnecessários se o nó mais próximo que será expandido for um beco sem saída, e se o algoritmo não detectar becos sem saída a solução pode nunca ser encontrada, oscilando entre dois nós próximos do objetivo, mas ambos sem saída.

Agora será apresentado uma busca com informação e exploração completa e ótima.

4.3.2 Busca A*

Uma vez que a enorme quantidade de vértices a serem pesquisados impedem que sejam usados algoritmos de busca exaustiva, como o algoritmo de busca em largura, é necessário uma estratégia que diminua a quantidades de nós pesquisados.

Dessa forma, o algoritmo A* se apresenta como a solução mais apropriada ao problema de Busca de Caminhos, pois encontra o caminho de menor custo de um vértice a outro examinando apenas os vizinhos mais promissores do vértice atual da busca. Segundo (PATEL, 2007) a busca A* é a melhor escolha na maioria dos casos.

O algoritmo A* é otimamente eficiente para qualquer função heurística dada, ou seja, nenhum outro algoritmo ótimo tem a garantia de expandir um número de nós menor que ele usando a mesma heurística. Os nós são avaliados de acordo com a fórmula:

$$f(n) = g(n) + h(n) \tag{4.1}$$

Onde *g(n)* corresponde ao custo exato do caminho desde o nó inicial até o nó n e *h(n)* o custo estimado do caminho de custo mais baixo para ir do nó n até o objetivo. Então pode-se afirmar que *f(n)* é o custo estimado da solução de custo mais baixo passando por n (RUSSEL; NORVING, 2004).

Desse modo pega-se o nó com menor valor $g(n) + h(n)$ para explorar, (Fig. 12), sendo este valor armazenado numa estrutura de nós já pesquisados: a lista de nós fechados; e seus filhos numa estrutura de nós a pesquisar: a lista de nós abertos; e desde que a função heurística satisfaça certas condições, a busca A* será completa e ótima. O símbolo '*' neste trabalho representa otimalidade.

Temos que *g(n)* e *h(n)* devem estar na mesma escala. Por exemplo, se *g(n)* é medido em horas e *h(n)* em metros, então o A* vai considerar g ou h muito grande se comparados, e não teremos bons caminhos ou a busca demorará mais do que deveria. Além da otimalidade não ser preservada (PATEL, 2007).

Mesmo com todas as vantagens da busca A*, o crescimento exponencial ocorrerá, a menos que o erro na função heurística não cresça com maior rapidez que o logaritmo do custo do caminho real. A condição para crescimento subexponencial em notação

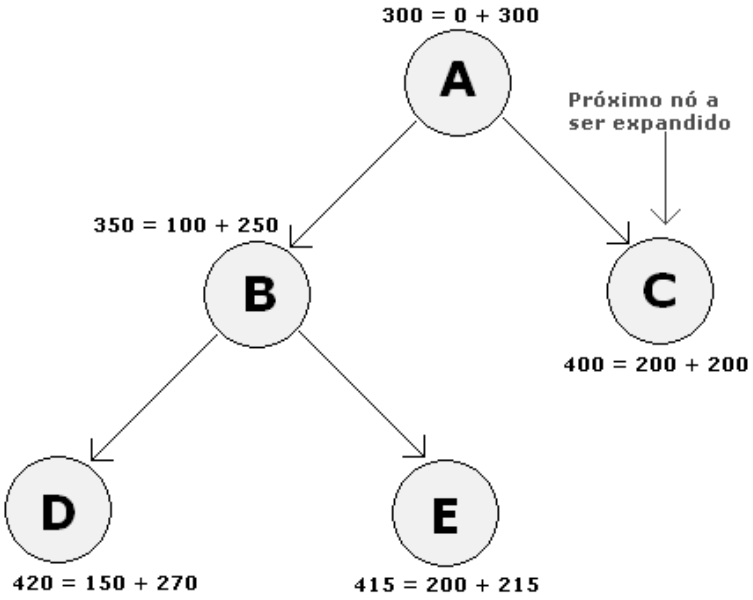

Figura 12: Esquema de uma expansão de nó em uma busca A*

matemática é:

$$|h(n) - h^*(n)| \leq O(log\ h^*(n)) \tag{4.2}$$

4.3.3 Heurística Admissível

A busca A* será ótima em uma busca em árvore se *h(n)* for admissível. A admissibilidade está no fato de *h(n)* nunca superestimar o custo para alcançar o objetivo. Tendo que *g(n)* é o custo exato para alcançar n, tem-se que *f(n)* nunca irá superestimar o custo verdadeiro de uma solução que passa por n. Propriedade da admissibilidade da heurística:

$$h(n) \leq h^*(n) \tag{4.3}$$

Onde $h^*(n)$ é o custo real do melhor dos caminhos. Um bom exemplo de uma heurística admissível é a distância euclidiana, pois é o menor caminho entre dois pontos, jamais superestimando seu custo verdadeiro.

Com uma heurística perfeita onde $h(n) = h^*(n)$, para todo estado n do caminho solução, teremos $f(n) = f^*(n)$, ou seja, expande somente os estados que formam o caminho solução. A pior heurística acontece quando $h(n) = 0$ caindo numa busca pela melhor escolha onde $f(n) = g(n)$. Uma análise disso leva a seguinte afirmação: Quanto maior for o valor da componente heurística, mais informativa e melhor ela é, gerando menos estados

no processo de busca, porém, tendo como limite o custo real do melhor dos caminhos.

Agora será discutido porque o A* usando busca em árvore é ótima se *h(n)* é admissível. Se m é um nó objetivo não-ótimo com $h(m) = 0$ e considerando C^* o custo da solução ótima é correto afirmar que:

$$f(m) = g(m) + h(m) > C^* \tag{4.4}$$

Considerando agora um nó que está em um caminho ótimo, se há solução esse nó existe, tem-se então:

$$f(n) = g(n) + h(n) \leq C^* \tag{4.5}$$

Mostra-se então que $f(n) \leq C^* < f(m)$. Em outras palavras: existe um caminho que ainda não foi escolhido melhor do que o caminho achado até o momento, e assim m, não será expandido e a busca A* continuará e retornará uma solução ótima.

4.3.4 Heurística Consistente

Se em vez da busca em árvore for usado busca em grafo, a prova mostrada anteriormente é derrubada, uma vez que soluções não-ótimas podem ser retornadas para um estado repetido, descartando o estado ótimo se ele não for o primeiro caminho gerado.

Existem duas maneiras de corrigir esse problema. A primeira solução é descartar o mais dispendioso entre dois caminhos quaisquer descobertos para o mesmo nó. A segunda é assegurar que o caminho ótimo para qualquer estado repetido é o primeiro a ser seguido. Para isso ser válido é preciso impor um requisito extra sobre *h(n)*, o requisito da consistência (RUSSEL; NORVING, 2004).

Uma heurística é consistente se para todos os nós:

$$h(n) \leq c(n, a, n') + h(n') \tag{4.6}$$

Onde para todo nó n, e todo sucessor n' de n gerado por qualquer ação a, o custo estimado de alcançar o objetivo a partir de n não é maior que o custo do passo de se

chegar a *n'* somado ao custo estimado de alcançar o objetivo a partir de *n*.

Como temos esse requisito sobre *h(n)* e o mesmo é uma restrição dos requisitos para uma heurística admissível, temos que toda heurística consistente é admissível, porém nem toda heurística admissível é consistente.

Uma heurística consistente pode ser vista como uma forma de desigualdade de triângulos geral, onde um lado não pode ser maior que a soma dos outros dois. A Figura 13 mostra essa relação no plano.

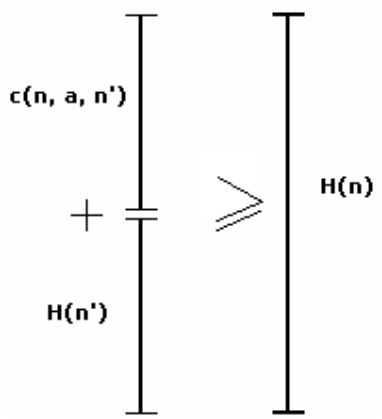

Figura 13: Condição necessária e suficiente para uma heurística consistente

Como característica da consistência temos: se *h(n)* é consistente, então os valores de *f(n)* ao longo de qualquer caminho são não-decrescentes, garantindo a otimalidade da solução. A prova deste fato é simples, seja *n'* um sucessor de *n*, então $g(n') = g(n) + c(n, a, n')$ para algum a, como consequência temos:

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n) \qquad (4.7)$$

Consequentemente o primeiro nó objetivo selecionado para expansão é ótimo, pois os outros nós são pelo menos tão dispendiosos quanto ele.

Porém, a maioria das heurísticas consistentes não atende a condição para crescimento subexponencial. Por essa razão, com frequência é impraticável insistir em uma solução ótima (RUSSEL; NORVING, 2004). É possível usar variantes da busca A* que encontrem rapidamente soluções não-ótimas, ou projetar heurísticas mais precisas, embora não estritamente admissíveis.

4.3.5 Criação de Heurísticas

Para explicar processo de criação de funções heurísticas admissíveis vamos tomar como exemplo o quebra-cabeça de oito peças (Fig. 14).

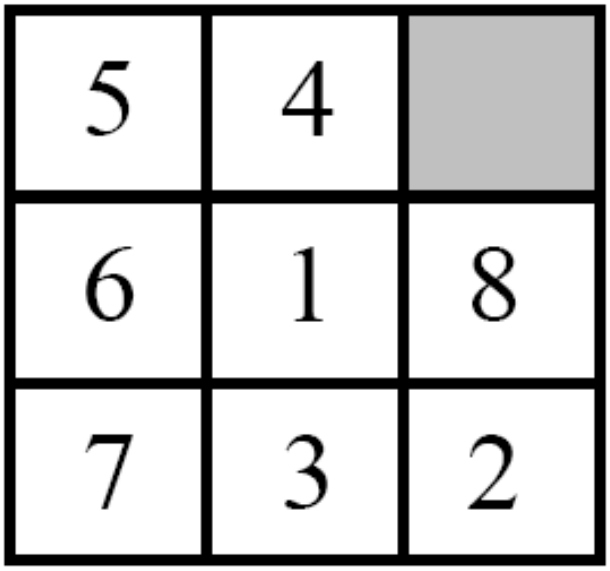

Figura 14: Quebra-cabeça de oito peças

Uma peça pode se mover somente se cumprir a seguinte restrição:

- Uma peça pode mover-se de A para B se A é adjacente a B, e B está vazio;

Um problema com menos restrições nos operadores diz-se relaxado. Muitas vezes o custo de uma solução exata para um problema relaxado é uma boa heurística para o problema original.

É possível gerarmos três problemas relaxados removendo uma ou ambas as condições do problema original. É mostrado agora algumas opções de relaxação do quebra-cabeça de oito peças:

- Uma peça pode mover-se de A para B se A é adjacente a B;
- Uma peça pode mover-se de A para B se B está vazio.;
- Uma peça pode mover-se de A para B;

O custo de uma solução ótima para um problema relaxado é uma heurística admissível para o problema original. (RUSSEL; NORVING, 2004)

Em um sistema cartesiano podemos definir a distância de Manhattan entre dois pontos num espaço euclidiano como a soma dos comprimentos da projeção da linha que une os pontos com os eixos das coordenadas. Por exemplo, num plano que contem os pontos P1 e P2, respectivamente com as coordenadas $(x1, y1)$ e $(x2, y2)$, é definido por: $|x_1 - x_2| + |y_1 - y_2|$.

A distância *Manhattan* pode ser derivada da seguinte relaxação: Uma peça pode mover-se de A para B se A é adjacente a B. Apresentando-se como uma boa heurística para o quebra-cabeça de oito peças, pois seu valor é sempre maior que a heurística dos outros dois problemas relaxados.

Se o problema relaxado for muito difícil de resolver, será dispendioso obter os valores da heurística correspondente.

4.4 Variantes do A*

Aqui são discutidos 3 variantes do Algoritmo A*: com busca bidirecional, com busca radial e com memória limitada.

4.4.1 Busca Bidirecional

A busca bidirecional encontra solução quando as duas frentes se interceptam, ou seja, possuem um vértice em comum, porém, para encontrar a solução ótima é necessário uma função de avaliação. Sendo s o nó da busca direta e t o da busca inversa:

$$fs(x) = gs(x) + hs(x)$$

$$ft(y) = gt(x') + ht(y)$$

onde *gs(x)* = custo do caminho da raiz s até o vértice x e *hs(x)* = custo estimado do vértice x até o objetivo t. Então sendo A o conjunto de vértices e *d(x,y)* a soma dos custos de um vértice x até um vértice y:

$$hs(x) = Min_{y \in As}\{d(x, y) + gt(y)\}$$

Com isso temos um valor heurístico mais preciso. Da mesma forma tem-se: *gt(y)* = custo do caminho da raiz s até o vértice y e *ht(y)* = custo estimado do vértice y até o

objetivo s. Então:

$$hs(x) = Min_{x \in As}\{d(y,x) + gs(x)\}$$

Dessa forma sabemos que $x = y$ quando $d(x,y) = 0$ ou $d(y,x) = 0$. E o caminho será ótimo se um vértice de interseção fornece o menor valor para todas as possibilidades.

A Busca Bidirecional apresenta as seguintes limitações: tem que existir um sistema inverso e apesar dos valores da componente heurística tornarem-se mais precisos, seu cálculo torna-se mais dispendioso, pois a heurística deve ser calculada para cada nó da frente oposta toda vez que avaliamos um estado.

4.4.2 Busca Radial

Na busca radial (Fig. 15), as buscas não são feitas de forma simultânea. Primeiramente é realizada uma busca em largura com determinada profundidade para a determinação do perímetro. Em seguida é realizada uma outra busca, geralmente a busca A*, apenas no sentido direto, ou seja, da raiz para o perímetro:

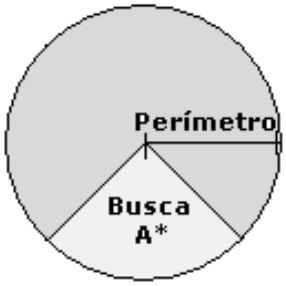

Figura 15: Busca radial com a busca A*

$$fs(x) = gs(x) + hs(x)$$

onde *gs(x)* = custo do caminho da raiz s até o vértice x e *hs(x)* = custo estimado do vértice x até o objetivo t. Então se P é o conjunto de vértices:

$$hs(x) = Min_{y \in P}\{d(x,y) + gt(y)\}$$

Se o raio for uniforme tem-se:

$$gt(y) = \rho \; \forall y \in P$$

$$hs(x) = Min_{y \in P}\{d(x,y)\} + \rho$$

Da mesma forma tem-se uma solução quando se encontra um vértice interseção, $x = y$ quando $d(x, y) = 0$. Esse caminho é ótimo se o vértice é o minorante da fórmula. Essa busca apresenta duas ressalvas: para gerar o perímetro necessita-se do sistema inverso e o tamanho do perímetro em quantidade de vértices depende do cálculo da heurística base.

4.4.3 Busca A* com memória limitada

Essa busca lida com a limitação de memória decorrente da superlotação da lista de abertos.

Enquanto a lista de nós abertos não atingir o tamanho máximo pré-estabelecido, o processo de busca funciona como o A* tradicional. Quando a lista de nós abertos atingir o tamanho máximo, compara-se o valor do estado a ser inserido com o pior. Se ele for melhor que o pior elimina-se o pior, caso contrário o mesmo não é inserido.

Há um problema nessa abordagem, perde-se a admissibilidade quando a quantidade de nós em abertos ultrapassa o tamanho máximo, porém é possível verificar a otimalidade da solução encontrada comparando o seu valor com o melhor dos piores. Se este valor for igual ou menor, a otimalidade foi preservada.

5 *Modelo Computacional*

Até aqui observamos a matemática necessária para que possamos fazer uma busca. Vimos os tipos de agentes, diferentes tipos de busca e o básico de cálculo de heurísticas. Agora vamos trazer esses modelos para o âmbito computacional, portanto veremos agora como computacionalmente soluciona-se o problema em questão.

As noções matemáticas vistas até aqui serão transportadas para o domínio discreto através de algoritmos diversos. Dentro do nosso contexto, quando esses algoritmos são usados na resolução de problemas por agentes, passam a ser conhecidos como algoritmos de planejamento.

Algoritmos de planejamento são aplicados em diversos problemas: humanos virtuais e robôs de humanóide, personagens de jogos de vídeo game, estacionar veículos, indústria aeroespacial, indústria automotiva. Em alguns casos, o trabalho progride de modelar, para algoritmos teóricos, e depois para software prático que é usado em indústria. "O futuro assegura tremenda excitação para os que participam do desenvolvimento de algoritmos de planejamento"(LAVALLE, 2006).

5.1 Agentes e Buscas

Aqui é apresentado o básico sobre agentes e buscas do ponto de vista computacional.

5.1.1 Agentes

Se mapeássemos todos os estados possíveis de um agente, teríamos uma tabela muito grande na maioria dos casos, ou até infinita se não fosse estabelecido um limite sobre o comprimento da sequência de ações a considerar. Devido a isso, usamos uma função para determinar o estado atual e o próximo estado do agente, essa **função do agente** é implementada por um **programa do agente**, essas duas idéias são distintas. *A função do*

agente é uma descrição matemática abstrata; o programa do agente é uma implementação, relacionada à arquitetura do agente.

O programa do agente implementa a função do agente mapeando percepções e escolhendo ações. Um agente pode ser representado como a soma de seu programa mais a sua arquitetura, *a arquitetura de um agente pode ser definida como algum dispositivo de computação com sensores e atuadores.* A Figura 16 mostra dois tipos de arquitetura para o mesmo programa de agente. O primeiro está inserido num ambiente virtual, sua arquitetura é o *hardware* do computador que simula os sensores e os atuadores. O segundo está no mundo real, sua arquitetura pode ser vista como todo o corpo do robô (LAVALLE, 2006).

Figura 16: Duas arquiteturas distintas para o mesmo programa de agente (LAVALLE, 2006).

5.1.2 Avaliação das Buscas

A busca escolhida para se implementar neste trabalho foi o A*. Foi escolhida porque é uma busca informada, ou seja, expande menos nós que uma busca sem informação se a função de avaliação for adequada ao problema. Temos também que nenhum outro algoritmo ótimo tem a garantia de expandir um número de nós menor que ele usando a mesma heurística, ou seja, é otimamente eficiente. Além disso, é própria para o tipo de agente em questão: **o agente baseado em objetivos**.

Temos então que ela gasta menos memória e executa mais rapidamente que as outras buscas, na maioria dos casos. É apresentado na Figura 17 uma tabela, cujo os dados são de (RUSSEL; NORVING, 2004), que compara a quantidade de nós abertos em três tipos diferentes de busca usados para solucionar um quebra-cabeça de oito peças: por aprofun-

damento iterativo sem informação, e duas buscas A* com heurísticas admissíveis diferentes, onde *h2* é mais próximo da distância real que *h1*, ou seja, $h2 > h1$. E d representa a profundidade da solução mais rasa e BAI é a abreviação de busca por aprofundamento iterativo.

d	BAI	*A*(h1)*	*A*(h2)*
2	10	6	6
4	112	13	12
6	680	20	18
8	6384	39	25
10	47127	93	39
12	3644035	227	73
14	...	539	113
16	...	1301	211
18	...	3056	363
20	...	7276	676
22	...	18094	1219
24	...	39135	1641

Figura 17: Comparação entre os custos das buscas

5.2 Modelo de Classes

A linguagem de programação utilizada foi a linguagem C++, sendo utilizado tanto a programação orientada a objetos quanto a programação estruturada, de acordo com a necessidade.

A classe onde efetivamente ocorre a busca A* chama-se *gcgASTAR*, é nela que estão implementados os métodos para se escolher o melhor nó e calcular o custo total, as estruturas para guardar os nós a pesquisar e os nós pesquisados, e as variantes da busca A*.

5.2.1 Relacionamentos da *gcgASTAR*

Com as classes estruturadas de acordo com o diagrama de classes da Figura 18 temos a possibilidade de usar o mesmo processo de busca para diferentes ambientes, ou seja, não é preciso mudar o código da busca se mudarmos o ambiente de trabalho. Além disso os agentes também se beneficiam dessa estrutura por não precisarem de múltiplas instâncias da classe *gcgASTAR* durante o processo de busca.

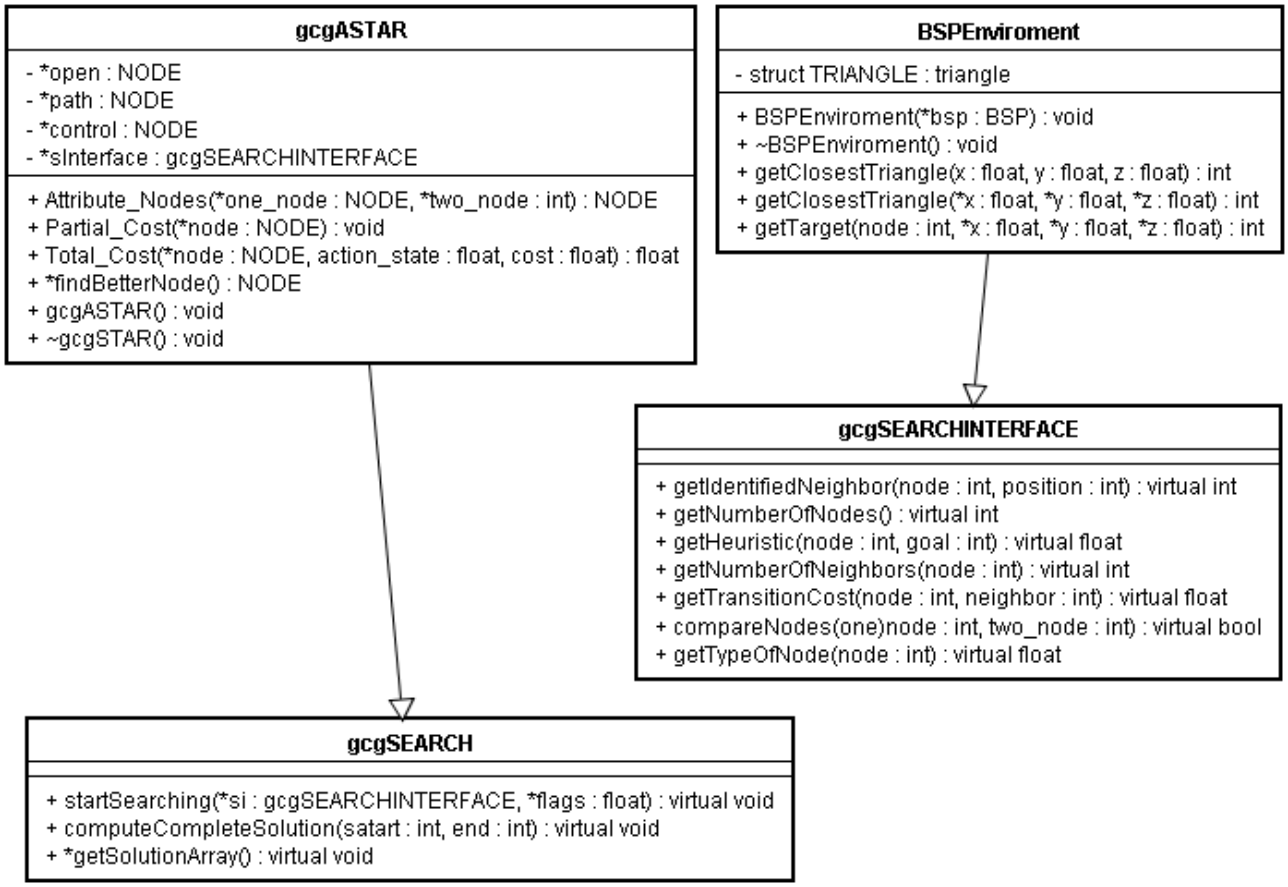

Figura 18: Diagrama de Classes do relacionamento da classe *gcgASTAR*

Para atuar conjuntamente com a classe *gcgASTAR* foram criadas duas classes abstratas: *gcgSEARCHINTERFACE*, para servir de interface com o ambiente, e a *gcgSEARCH*, para comunicar-se com o agente.

Essa comunicação da classe *gcgSEARCHINTERFACE* com o ambiente é feita da seguinte forma: a classe responsável pelos métodos referentes ao ambiente, tais como definição da geometria do ambiente e distâncias dentro do mesmo, é a classe *BSPEnvironment*. Dessa forma ela herda a classe *gcgSEARCHINTERFACE*, sendo obrigada a implementar todos os seus métodos.

Dentro da classe *gcgSEARCH* há um ponteiro para a classe *gcgSEARCHINTERFACE*, temos então sua instância, através dessa instância a classe *gcgASTAR* pode chamar os métodos da classe *gcgSEARCHINTERFACE*, ou seja, quando necessário podemos obter respostas da geometria do ambiente através dessa interface.

A classe *gcgASTAR* herda a classe abstrata *gcgSEARCH*, ou seja, toda busca implementada deverá conter pelo menos os métodos dessa classe, esses métodos englobam o básico que toda busca deve ter.

5.3 Processo de Busca na Classe *gcgASTAR*

Como a busca A* é feita examinando quais nós expandir, temos que criar uma estrutura para guardar as informações dos nós, para isso foi criado inicialmente um registro chamado *NODE* com quatro campos:

- *identified*: o identificador do nó;
- *partial_cost*: Custo do nó inicial ao nó atual, ou seja, $g(n)$;
- *total_cost*: Soma da heurística com o *partial_cost*, ou seja, armazena o resultado da função de avaliação $f(n) = g(n) + h(n)$, onde n é um nó;
- *father_node*: Nó que o expandiu.

O processo de busca da classe *gcgASTAR* começa com o método *startSearching* onde são inicializados três vetores de nós: um representando a lista de abertos, outro a lista de fechados e outro o caminho escolhido. Para alocar esses vetores eficientemente é feito um cálculo com a quantidade de nós total do ambiente, que é recuperada pela interface *gcgSEARCHINTERFACE*, quanto mais nós no ambiente, mais espaço é alocado. Aqui também são inicializadas outras variáveis globais, como controle de realocação entre outras.

Em seguida é chamado o método *computeCompleteSolution*, onde o cálculo do melhor caminho é feito. Ele recebe o nó inicial, alocado na lista de abertos, e o nó destino, em seguida os compara, se forem iguais a busca termina, porém se forem diferentes esse nó precisa ser expandido. Desse modo ele é posto na lista de fechados e expandido, e seus filhos inseridos na lista de abertos, depois pesquisa-se essa lista afim de se encontrar o melhor nó. As diferenças no modo que esse melhor nó é escolhido é o que difere as variantes do A*, ou seja, na maioria dos casos só é preciso mudar essa função para termos uma outra variante do A*. Cada uma das variantes implementadas serão descritas a parte na seção 5.4.

Depois de escolhido o melhor nó é verificado se o mesmo corresponde ao nó objetivo, em caso negativo o nó é expandido e o processo é repetido, em caso afirmativo a busca termina e o caminho a ser seguido pode ser descoberto retroativamente. Pega-se o nó pai do nó em questão, e depois o nó pai de seu nó pai, e assim sucessivamente, sempre gravando-os na lista de nós do caminho a ser seguido, até o pai ser o próprio nó inicial. Precisa ser guardado também a quantidade de nós que esse caminho possui.

O caminho é repassado para o agente através do método *getSolutionArray*.

5.4 Seleção do Melhor Nó

Aqui é explicado como é escolhido o melhor nó da lista de abertos a fim de expandi-lo e continuar com o processo de busca. Existem diferentes formas de se implementar a escolha do melhor nó. Algo importante a considerar é o gasto de memória e o gasto de tempo de cada tipo de implementação. Segundo (RUSSEL; NORVING, 2004), embora não exista nenhuma teoria para explicar a relação inversamente proporcional entre tempo e memória, este parece ser um problema inevitável.

5.4.1 Calculando o Custo Parcial: *g(n)* e a Heurística: *h(n)*

Para calcular o custo parcial pegamos o custo parcial do nó pai e somamos com o custo real do nó pai até o nó atual, para efetuar este cálculo precisamos do auxílio da interface *gcgSEARCHINTERFACE*, pois através dos dados que o ambiente nos retornar é que será calculado essa distância.

A heurística é calculada também através de um método da interface, calcula-se a distância euclidiana do nó atual até o nó objetivo, já foi demonstrado na seção 4.3, subseções 4.3.3 e 4.3.4 que esta heurística é admissível e consistente.

5.4.2 Inserção e Remoção na Lista de Abertos

O primeiro nó a ser escolhido como melhor nó é o próprio nó inicial. Daí pega-se todos os seus nós vizinhos através da interface *gcgSEARCHINTERFACE*, e os mesmos são colocados na lista de abertos, e para cada um é calculado o custo parcial, que no caso do nó inicial é zero. Calcula-se a heurística *h(n)* e somando-os temos o custo total, o *f(n)*. Em seguida é feita uma pesquisa na lista de abertos para achar o nó com menor custo, o mesmo é retornado, e o processo reinicia.

A função base para a escolha desse melhor nó chama-se *findBetterNode()* e foi criada dentro da classe *gcgASTAR*.

Foram criadas três classes adicionais, uma para cada maneira de se escolher o melhor nó, que herdam da classe *gcgASTAR* e modificam somente a implementação da função padrão *findBetterNode()* da mesma, com o intuito de abstrair o cálculo dessa escolha e

modularizar suas diferentes implementações, e isso foi feito de acordo com a Figura 19.

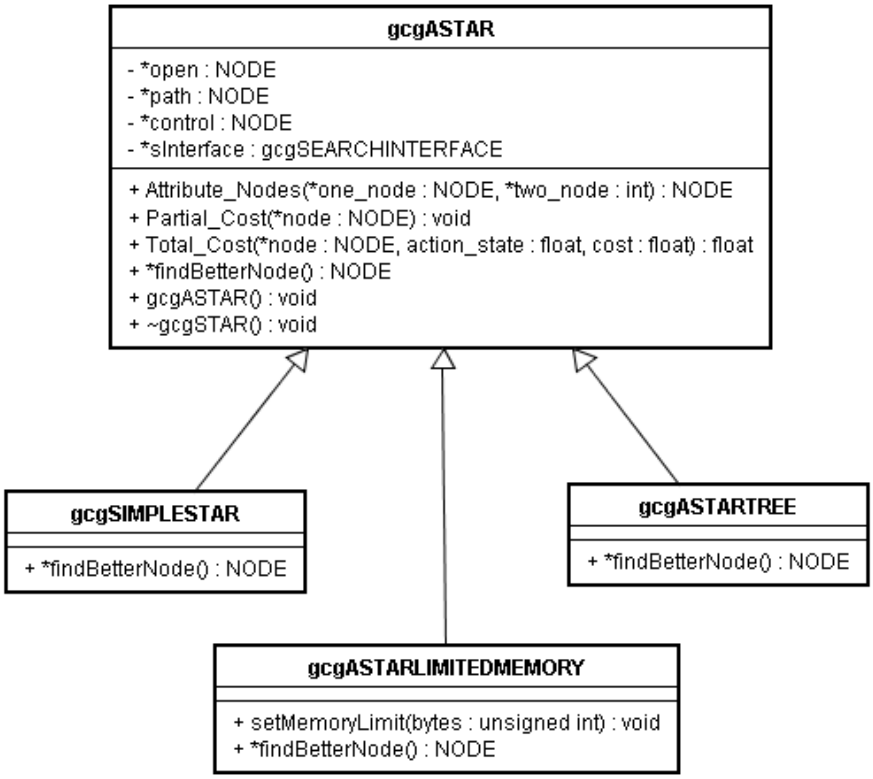

Figura 19: Diagrama de Classes do relacionamento da escolha do melhor nó com a classe *gcgASTAR*

Temos portanto quatro maneiras de se calcular o melhor nó neste trabalho, e todos usam o mesmo princípio descrito acima, com pequenas alterações entre os mesmos. A principal diferença entre eles está no gasto de tempo e memória.

Primeiramente foi criado uma classe chamada *gcgSIMPLEASTAR*, onde a inserção é feita inserindo-se o nó na última posição da lista, complexidade *O(1)*, e para efetuar a pesquisa tinha-se que percorrer toda a lista, *O(m)*, onde m é a quantidade de nós em abertos. Apesar da inserção ser rápida, a medida que ambientes maiores foram sendo testados, a lista de abertos cresceu muito, tornando o programa lento. Temos também o gasto de memória, pois todos os nós em abertos precisam ser armazenados. O nó expandido recebe um custo elevado para não mais ser escolhido durante a busca pelo melhor nó.

A Segunda classe criada chama-se *gcgASTARTREE*, e usou o conceito de árvore, a mesma está estruturada de forma a manter os nós mais custosos a direita. Para isso foram acrescentados dois novo campos na estrutura *NODE*:

- *left_node*: um ponteiro para representar o nó a esquerda;

- *right_node*: um ponteiro para representar o nó a direita.

A principal mudança foi acrescentar o cálculo de onde o novo nó deve entrar na árvore, porém esse cálculo só trabalha com ponteiros, deixando-o bem veloz. Temos então duas estruturas vinculadas: uma lista, e uma árvore binária montada por ponteiros a partir dessa lista. A grande vantagem dessa abordagem está no fato de que para pesquisar o melhor nó na lista de abertos, basta buscar o nó mais a esquerda da árvore. Temos então que a ordem de complexidade é $\ominus(log\ m)$ no caso médio e *O(m)* no pior caso, quando a árvore está desbalanceada. Depois que um nó é expandido ele não pode mais ser incluído na pesquisa pelo melhor nó, neste caso cortamos suas ligações de ponteiros na árvore e rearrumamos a árvore.

O terceiro método chama-se simplesmente *findBetterNode()*, e por ser o padrão foi implementado dentro da própria classe *gcgASTAR*, ela possui a seguinte vantagem adicional: como o ambiente que lhe é passado pode ser representado por um grafo, conseguimos diminuir a quantidade de nós na lista de abertos, conforme explicado abaixo.

Agora nossa lista de nós fechados será muito útil, antes do nó ser inserido na lista é feita a verificação se o mesmo já está na lista de fechados, se estiver ele não é adicionado, caso contrário adiciona. Apesar dessa verificação demandar tempo, a lista de nós fechados é muito menor que a lista de nós abertos, por exemplo, se cada nó tiver em torno de cinco vizinhos, a lista de fechados tende a ser cinco vezes menor. Não interferindo de maneira significativa na complexidade de tempo. Por outro lado, economiza-se uma boa quantidade de espaço procurando estados repetidos durante a busca e não adicionando-os na lista de abertos, ou seja, elimina a possibilidade de visita duplicada. Segundo os testes realizados, além de economizarmos memória com essa verificação na lista de fechados, acabamos por economizar tempo também, a lista de abertos fica tão menor que economizamos bem mais tempo com a pesquisa na lista de abertos do que é gasto na comparação com a lista de fechados. A Tabela 1 demonstra esses resultados.

Tabela 1: Tabela de comparação da quantidade de nós inseridos na lista de abertos

Nó Origem	Nó Destino	Nós do Caminho	Nós Abertos-Grafo	Nós Abertos-Árvore
17	41	10	35	90
16	17	2	4	4
17	107	10	198	mais que 128.344

Ilustrando o que foi mencionado na subseção 4.2.5 sobre o aumento no espaço de estados tendo a Tabela 1 como fator de comparação: no grafo o espaço de estados possui

$2d^2$ folhas, ao contrário da árvore que possui 4^d folhas. Tomando como base a primeira linha da tabela com 35 nós abertos no Grafo, temos que o fator d é próximo de $2 * d^2 = 35 \Rightarrow d = 4.18$, então na árvore teríamos no máximo por volta de $4^{4.18} = 328$. Já a terceira linha nos mostra uma disparidade muito grande quando a busca necessita de mais nós na lista de abertos, temos 198 nós abertos na mesma, então $2 * d^2 = 198$, temos que d é aproximadamente 9.9, então passando esse valor pra busca em árvore poderíamos ter até $4^{9.9} = 912838$ estados pesquisados, caracterizando um aumento bem elevado, e quanto mais se aumenta a necessidade de mais nós na lista de abertos, maior fica essa discrepância.

Como agora estamos levando em consideração um grafo, a heurística precisa ser consistente para a busca continuar ótima, a heurística usado nesta implementação é a distância euclidiana, ou seja, é consistente.

A terceira classe lida com a limitação de memória da máquina, quando o ambiente a considerar for muito grande pode ocorrer esse tipo de limitação. O nome dessa classe é *gcgASTARLIMITEDMEMORY*. Esta implementação está usando o mesmo esquema básico de busca em grafo citado acima, porém agora é feita uma verificação a mais. Se durante a execução do algoritmo o espaço armazenado superar um valor pré-estabelecido, compara-se o valor do estado a ser inserido com o pior. Se ele for melhor que o pior elimina-se o pior da lista de abertos, caso contrário o mesmo não é inserido na lista.

Para essa implementação foi necessário adicionar mais um campo na estrutura *NODE*:

- *position*: armazena um inteiro que representa sua posição na lista de abertos;

Essa inserção foi necessário para sabermos a posição do pior nó da árvore na lista de nós abertos, podendo assim substituí-lo sem precisar pesquisar toda a lista de abertos.

O otimalidade é verificada comparando o nó com o melhor dos piores. Se este valor for igual ou menor, a otimalidade foi preservada. Porém, neste método, não restringimos o problema somente a solução ótima, ou seja, há um aviso se a otimalidade foi preservada ou não durante a busca. Ela não é interrompida.

Apesar das vantagens desse método há um empecilho, depois que a memória encontra-se cheia, a troca de nós na lista de abertos passa a ser constante, tornando o programa muito lento. Portanto foi escolhido como método padrão a busca em grafo, devido as suas vantagens com relação aos outros métodos aqui apresentados, conforme mostrado na Tabela 2, de modo que quando é gasto toda a memória ele considera que não há possibilidade de caminho e determina um novo objetivo.

Tabela 2: Tabela de comparação entre os métodos de escolha do melhor nó da implementação

	SIMPLE	TREE	GRAFO	LIMITEDMEMORY
Comp.(Tempo)	O(n)	$\ominus(log\ m)$	$\ominus(log\ m)$	$\ominus(log\ m)$
Comp.(Espaço)	4^d	4^d	$2d^2$	$2d^2$
Vantagem	Simplicidade	Vel.	Vel. e Espaço	Memória
Desvantagem	Complexidades	Espaço	Memória	+Lento

6 Aplicação

Neste capítulo aplicamos os conceitos já vistos em um problema específico de buscas em ambientes 3D, o mesmo sendo solucionado por um agente autônomo. Criar movimentos realísticos para animações é um importante problema com diversas aplicações na indústria de jogos e animações. (LAU; KUFFNER, 2005)

6.1 O Ambiente Virtual

O Ambiente virtual onde a busca é feita possui particionamento binário do espaço, um método popular de representação hierárquica do cenário, no qual, um espaço n-dimensional é recursivamente particionado em subconjuntos convexos por hiperplanos. Uma árvore de partição binária do espaço é uma estrutura de dados usada para representar o particionamento. (LATTARI, 2007)

Posteriormente, através desse particionamento, o espaço onde a busca ocorrerá é dividido em triângulos, e cada centróide desses triângulos são ligados aos centróides dos triângulos vizinhos, formando um grafo, além disso a quantidade de triângulos no ambiente é controlável, portanto quanto mais triângulos em cena mais preciso tende a ser o caminho do agente, porém mais custoso será esse cálculo. Um modelo de ambiente com o grafo do caminho que pode ser seguido é mostrado na Figura 20, ande as linhas claras representam o grafo.

Todos os cálculos que envolvem conhecimento do ambiente são calculados através da Interface *gcgSEARCHINTERFACE*, tais como cálculo de distâncias usados nos métodos para se determinar heurísticas e custos, quantidade de vizinhos e número de nós do grafo. Dessa forma conseguimos encapsular o ambiente, ou seja, qualquer tipo de ambiente pode ser utilizado aqui desde que se enquadre no tipo de estrutura de particionamento já mencionado.

Parte do que é calculado precisa ser armazenado, então o valor é passado para os

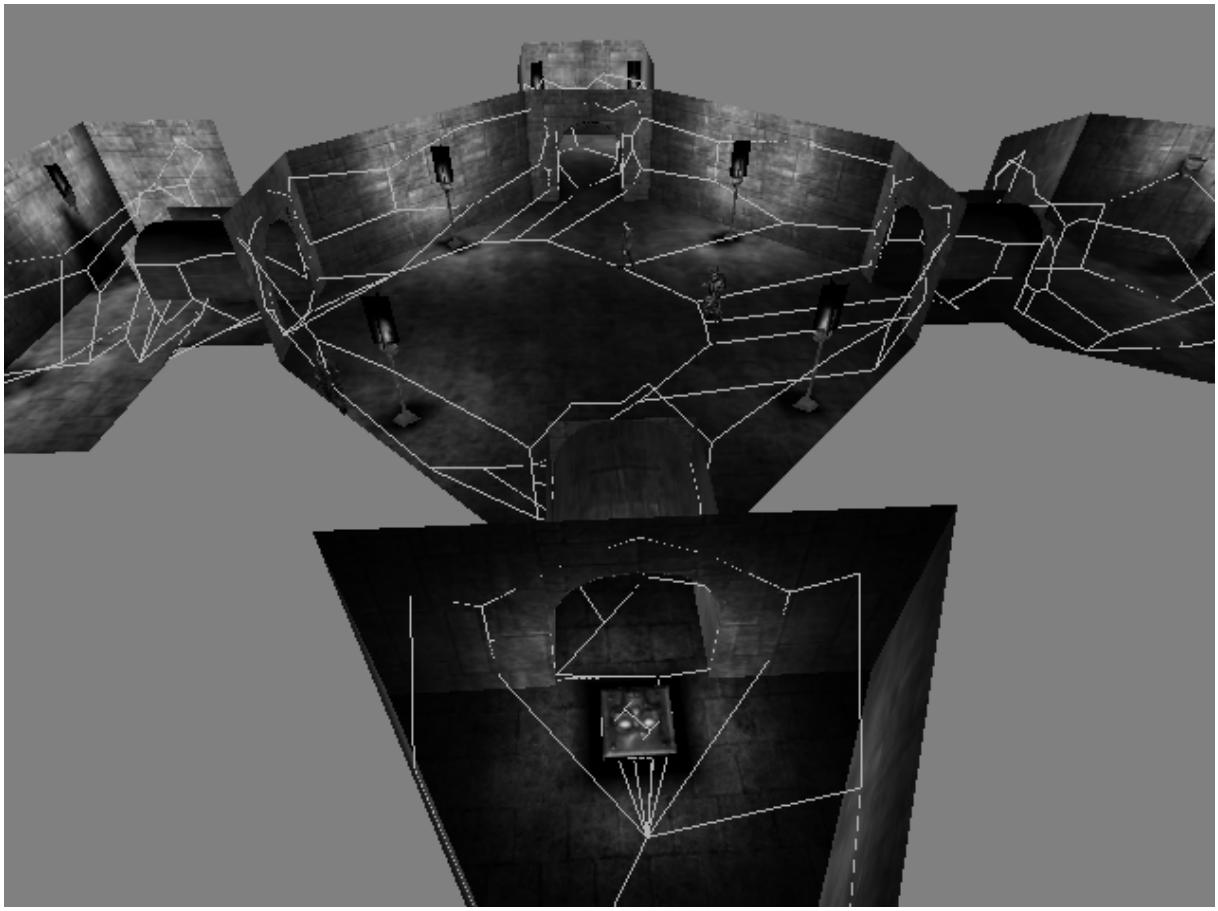

Figura 20: Exemplo de um dos ambientes usados com o grafo formado pela ligação dos nós

campos da estrutura *NODE*, como exemplo temos o custo total, que é a soma do custo parcial mais a heurística. Ou para alguma variável da classe *gcgASTAR*, como a variável *amount_of_nodes* usada para guardar quantos nós existem no ambiente.

6.2 Modelo do Agente

Aqui é discutida a abstração do agente usado no processo e execução da busca no ambiente 3D. O agente recebe as informações do ambiente através de sensores e executa ações dentro desse ambiente através de atuadores.

Os sensores do agente estão implementados na estrutura *SENSORS*, esta estrutura guarda as seguintes informações: posição atual do agente e geometria do ambiente onde se encontra, podendo futuramente ser limitada a algum tipo de raio de visão, sem modificação em sua estrutura ou na do agente.

A ação que o agente deve tomar é determinado por uma máquina de estados, imple-

mentada na própria classe do *Agent*, explicada na subseção 6.3.1. Essa ação é representada por uma estrutura contendo o código da ação a ser tomada, o estado dessa ação e o próximo alvo. Nesta implementação as seguintes ações são possíveis:

- *STOPPED_STAND*: o agente deve parar;
- *RUNNING_STAND*: o agente deve correr;
- *WALKING_STAND*: o agente deve andar;
- *STOPPED_CRAW*: o agente deve agachar;
- *WALKING_CRAW*: o agente deve andar agachado;
- *JUMP_UP*: o agente deve pular em pé;
- *JUMP_FRONT*: o agente deve pular para frente;
- *DYING*: o agente deve morrer;
- *DEAD*: o agente está morto, não pode executar ações.

6.2.1 Mente e Corpo

Foram criadas duas abstrações para o agente: **A mente e o Corpo**. Ambas são classes abstratas de modo que toda mente e corpo implementados devem possuir seus métodos básicos.

A mente necessita de um método que calcule qual ação executar em determinadas ocasiões, este método recebeu o nome de *chooseAction* e no caso de alguma ação não puder ser executada é chamado o método *revaluateWrongs*.

A classe *Body* é a responsável por captar a informação do ambiente através dos sensores e passar esta informação para a classe *Mind*, e depois que ela retorna o que fazer o *Body* executa essa ação, ou seja, o atuador é o próprio corpo do agente, ele também é responsável por retornar sua posição: função *returnPosition*, desenhar-se: função *Draw*, e inicializar sua posição no mapa: função*configPosition*.

Estruturado dessa forma, conforme ilustrado na Figura 21, **tanto o corpo quanto a mente tornam-se escaláveis para diferentes tipos de agentes**, podendo uma mesma mente ser usada por vários corpos diferentes, assim como uma mesma estrutura de corpo usar diferentes mentes. Onde *dt* representa o tempo passado no ambiente.

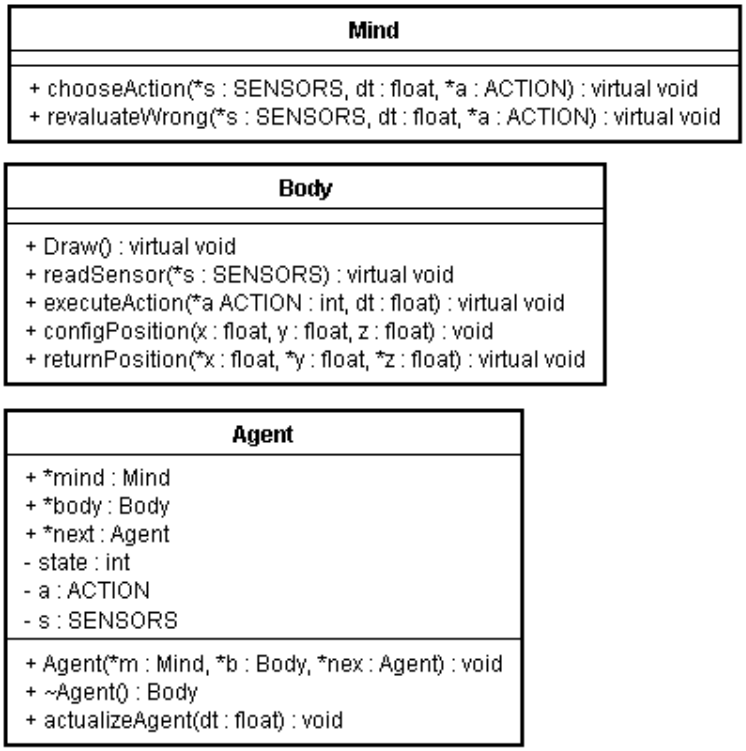

Figura 21: Diagrama das classes *Body*, *Mind* e *Agent*

6.3 A Classe *Agent*

A classe *Agent* possui uma instância da classe *Mind* e uma instância da classe *Body*. Também possui um ponteiro para o próximo agente, **podendo assim um mesmo ambiente possuir vários tipos de agentes diferentes**. Possui também a estrutura *SENSORS* e a estrutura *ACTION* e uma variável inteira para guardar o estado corrente do agente.

6.3.1 Máquina de Estados

O Comportamento do agente é regido por uma máquina de estados implementada em sua própria classe *Agent*, no método *actualizeAgent* (Fig. 22). Ela é responsável por dada uma ação e o estado atual do ambiente, captado pelos sensores, escolher qual a próxima ação que o agente deve executar.

Estados possíveis da maquina de estados finitos do agente:

- *INITIAL_STATE*: Estado que inicia a máquina de estados;
- *DECISION_STATE*: o agente deve escolher que ação tomar;

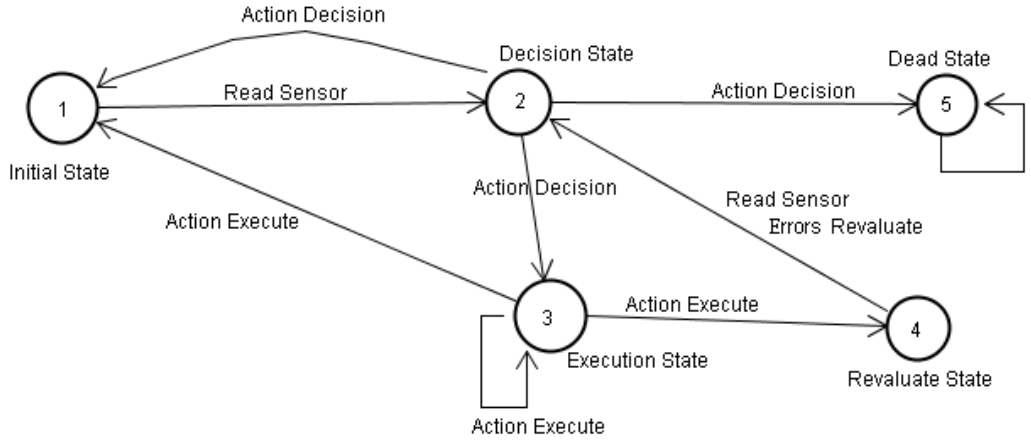

Figura 22: Máquina de estados que representa o comportamento do Agente

- *EXECUTION_STATE*: o agente deve executar a ação escolhida;
- *DEAD_STATE*: o agente morreu e não deve mais tomar nenhuma ação;
- *REVALUATE_STATE*: o agente avalia porque alguma ação não pode ser executada;

É necessário também determinar como está o andamento da ação que se está executando, para isso foram criados os *status* das ações. Código dos *status* das ações:

- *EXECUTED_STATUS*: é ativada quando a ação acaba de ser executada;
- *CURRENT_STATUS*: permanece ativa enquanto a ação ainda não terminou;
- *COLISIONERROR_STATUS*: é ativada quando ocorre alguma colisão entre o agente e o ambiente;
- *TIRED_STATUS*: *status* especial que determina que depois de algum tempo *dt* o agente deve descansar;

6.4 As Mentes do Agente

De modo a ilustrar a possibilidade de criação de múltiplas mentes tendo como base uma mente primitiva, foram implementados três tipos de mentes diferentes, cada uma dessas três classes implementam os métodos definidos na abstração da *Mind*. A Figura 23 mostra o mecanismo de herança utilizado entre elas.

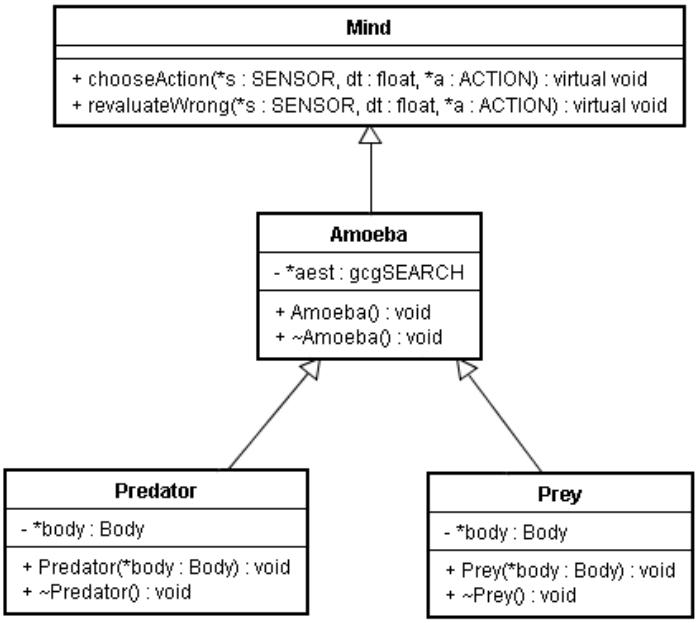

Figura 23: Mecanismo de herança utilizado nas implementações da classe *Mind*

6.4.1 Mente Ameba

A classe *Amoeba* é a mais simples e foi a primeira a ser implementada. Os nós origem e destino são escolhidos randomicamente e para calcular esse caminho é utilizado o algoritmo A* implementado na classe *gcgASTAR*.

A classe *Amoeba* possui uma instância da interface *gcgSEARCH*, ou seja, pode usar diferentes tipos de buscas, já que todas as buscas criadas precisam herdar da classe *gcgSEARCH*. Em nossa implementação a classe herdeira da mesma é a *gcgASTAR*.

Assim que é iniciado o método *chooseAction* é feito o seguinte: depois de um instante pré-estipulado de tempo *dt* é iniciado o processo mental do agente, esse tempo *dt* pode ser visto como o tempo de reação, ou seja, o tempo que o agente leva para começar a decidir que ação tomar.

O nó inicial recebe a posição onde o agente está no momento através da estrutura *SENSORS*. O nó final ou objetivo é escolhido de forma randômica utilizando os dados que os sensores tem do ambiente.

Logo após é iniciado o processo de busca do agente, para isso é criado um objeto *gcgSTAR* e a partir desse objeto a busca é iniciada e executada, no final da busca é retornado o caminho a ser seguido pela função *getSolutionArray*, e seu tamanho pela função *computeCompleteSolution*.

Por estarmos trabalhando num ambiente de coordenadas *float* é útil considerar válido um pequeno erro na posição, para isso utilizamos como auxiliar um valor pequeno pré-estabelecido como margem de erro, desse modo impedimos que o agente fique variando em torno de uma mesma posição procurando achar o valor exato esperado.

O código da ação recebe o estado *WALKING_STAND* em seguida o próximo alvo é captado pelo sensor e passado para a estrutura *ACTION*. Caso este passo seja o final, o código da ação recebe um comando para parar, cujo código é *STOPPED_STAND*.

6.4.2 Mente Predator

A classe *Predator* possui uma mente um pouco mais elaborada e herda da classe *Amoeba* cuja mente é primitiva. A principal mudança está no fato de seu construtor receber uma classe *Body* como parâmetro, que será perseguida pelo predador.

O cálculo do nó objetivo vai depender da localização do corpo perseguido. Desta forma é calculado a posição do corpo que é então captada pelos sensores.

O instante *dt* pré-estipulado é menor que o da mente *Amoeba* afim de dar uma vantagem ao predador, visto que ele deve calcular mais vezes o nó objetivo já que o corpo que está sendo perseguido muda continuamente de posição. O restante dos procedimentos são calculados como na mente *Amoeba*.

6.4.3 Mente Presa

A classe *Prey* também herda da classe *Amoeba*. Aqui também o construtor dessa classe recebe uma classe *Body* como parâmetro, porém neste caso a *Prey* toma ações que a levam a fugir desse corpo.

O cálculo do nó objetivo vai depender da localização do corpo que se deseja fugir. É calculado a posição do corpo que é então captada pelos sensores.

O instante *dt* pré-estipulado está entre os valores da mente *Amoeba* e da mente *Predator*, afim de dar uma vantagem a presa, mas deixando ainda uma brecha para que o predador possa pegá-lo. O restante dos procedimentos são calculados como na mente *Amoeba*.

6.5 Os Corpos do Agente

Inicialmente foram feitos corpos bem simples para os agentes, seu formato é o de uma bola, mas já implementando todos os métodos da classe abstrata *Body*. No construtor da classe *Ball* é passado a geometria do ambiente, os triângulos da cena e sua quantidade, e alguns parâmetros próprios da bola como cor e velocidade. A função *executeAction* é implementado de modo que o código da ação é verificado e executado de uma maneira diferente para cada código da ação, se a ação for executada com sucesso, o *status* da ação é definida como *EXECUTED_STATUS*.

Posteriormente foi criado um corpo bem estruturado, com movimentos elaborados. As extensões desses arquivos é *.mdl*, dessa maneira abstraímos a forma como o corpo é definido, agora vários corpos diferentes, desde que possuam a extensão *.mdl*, podem ser executados através da mesma classe *MDLBody*. Essa classe também implementa todas as funções da classe abstrata *Body*, porém seu construtor recebe somente a geometria do ambiente, os triângulos da cena e sua quantidade.

6.6 A Execução do Programa

Primeiramente todas as funções cujo processamento ocorre na placa de vídeo são inicializados, o ambiente onde se deseja fazer a busca é lido, e então começa-se o processo referente ao agente.

É criado um ou mais objetos da classe *Body* e em seguida suas posição iniciais são definidas. Então é criado um ou mais objetos da classe *Mind*, e caso seja predador ou presa é passado em seu construtor um dos corpos já criados.

Agora que os objetos mentes e corpos já foram criados, são passados uma mente e um corpo no construtor do agente, que recebe também um ponteiro para o próximo agente a ser criado, e então o processo continua até termos a quantidade de agentes que se deseja no ambiente.

Na função *simulation*, onde é calculado o tempo *dt* através da quantidade de quadros por segundo, também é feita a atualização dos agentes. Enquanto o próximo agente for diferente de vazio, a máquina de estados é inicializada através do método *actualizeAgent* da classe *Agent* que recebe como parâmetro a variável *dt*.

Durante a execução do programa temos os resultados ilustrados nas Figuras 24, 25, 26, 27, 28:

Figura 24: Execução do programa depois de 3 segundos

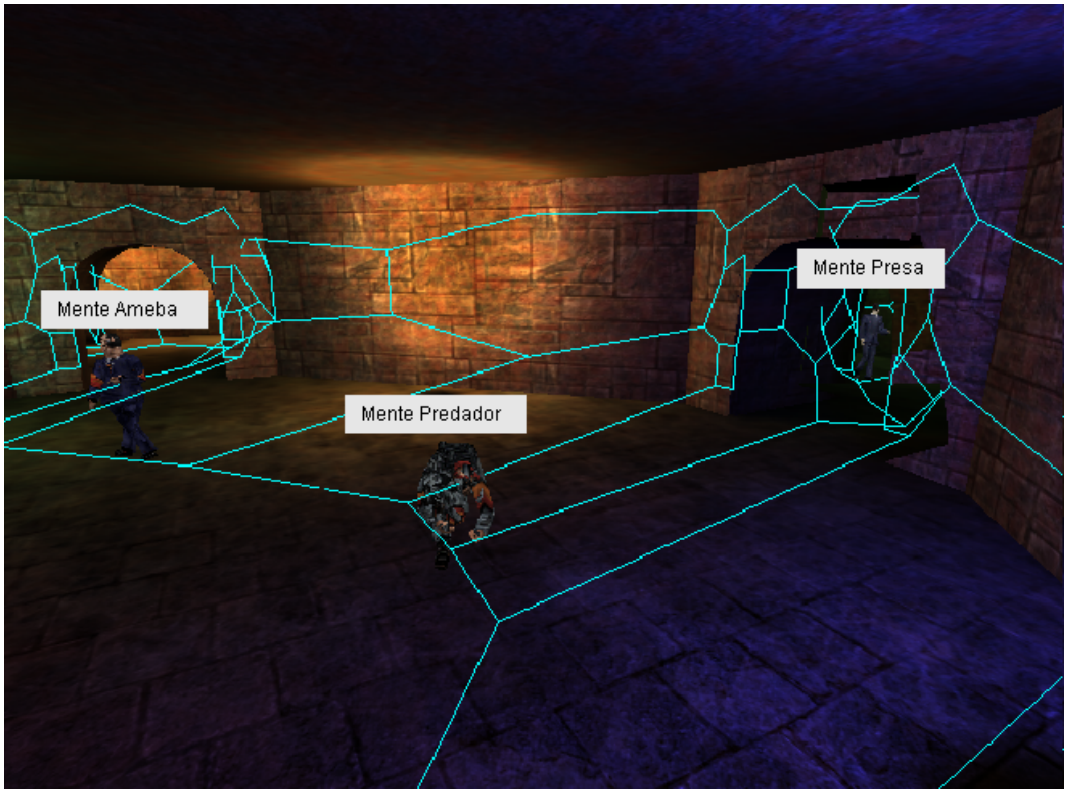

Figura 25: Execução do programa depois de 6 segundos

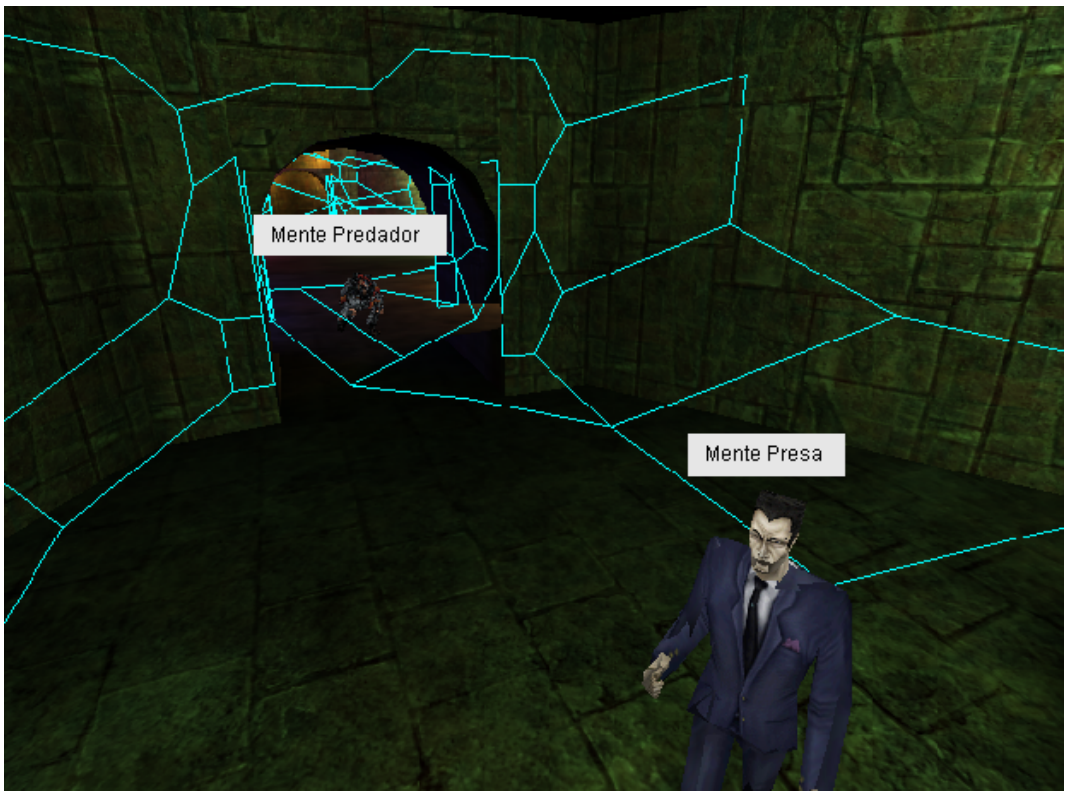

Figura 26: Execução do programa depois de 9 segundos

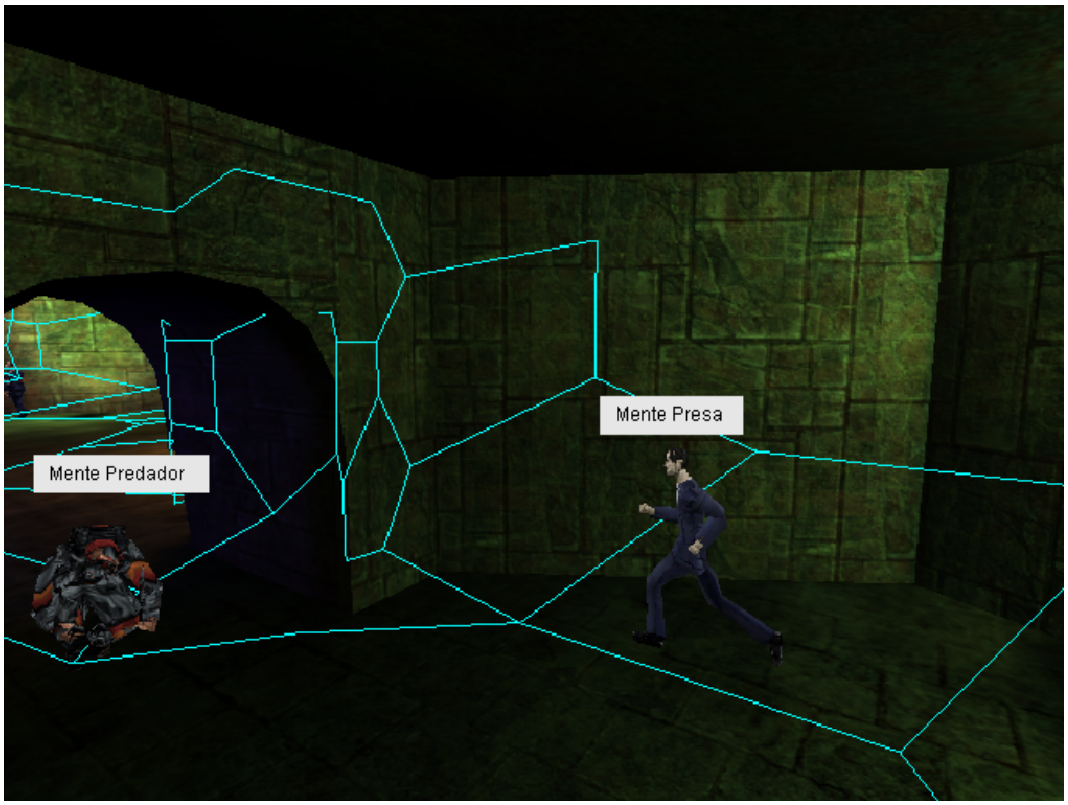

Figura 27: Execução do programa depois de 12 segundos

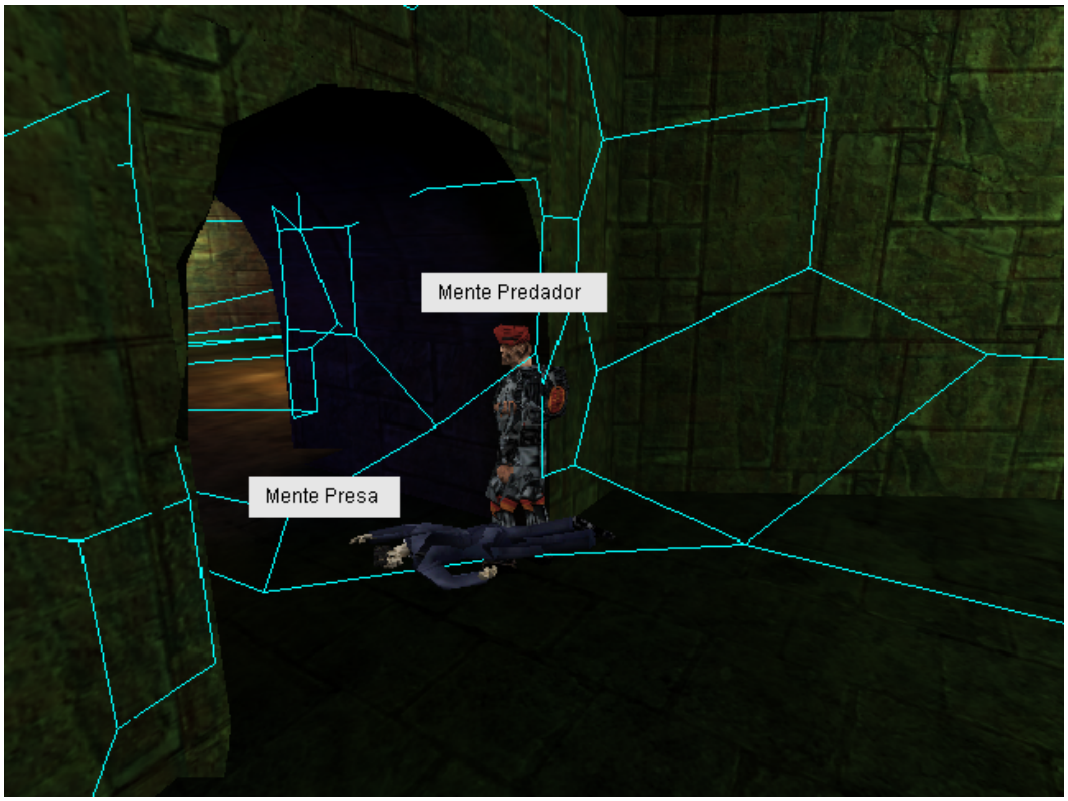

Figura 28: Execução do programa depois de 15 segundos

7 Conclusão

Neste trabalho foram vistos os conceitos e métodos necessários para que uma busca 3D, feita por um agente autônomo num ambiente complexo e dinâmico, fosse possível. Foram vistos diferentes tipos de agentes com ênfase no agente baseado em objetivos, o qual foi implementado. Diversas buscas foram abordadas e uma atenção especial foi dada ao algoritmo A*. Ele e algumas de suas variantes foram implementadas, vários testes foram realizados, e com base nos resultados foi montado uma tabela comparativa entre os mesmos. Depois de exaustivos testes chegamos à conclusão que a busca A* usando grafo é a melhor escolha para essa implementação.

O agente foi dividido em dois componentes básicos: corpo e mente, sendo os mesmos independentes entre si. O agente percebe o ambiente através de sensores que são captados pelo corpo, tais informações são processadas pela mente, o caminho é escolhido e essa informações são passadas para o corpo para o que ele atue nesse ambiente através de ações pré-definidas.

Corpos e mentes podem ser agrupados a fim de formar diversos tipos de agentes diferentes. Três tipos de mentes foram implementadas:*Ameba*, *Predador* e *Presa*, sendo que as mentes mais evoluídas: *Predador* e *Presa* herdam da classe *Ameba*, dessa maneira podemos fazer um esquema de evolução de mentes, onde mentes mais evoluídas englobam os princípios básicos das mentes primitivas através de herança. Já os corpos podem ser inúmeros através da mesma classe *MDLBody*, bastando para isso passar diferentes arquivos *.mdl* de personagem já montados. E caso haja necessidade de corpos com outros tipos de extensões, basta criar uma outra classe que herde de corpo.

Um mesmo ambiente pode conter vários agentes, cada qual com sua própria mente e corpo. As ações que o agente toma quando está percorrendo um caminho é definido por uma máquina de estados finitos dentro da classe *Agente*. Tal máquina de estados é responsável por dada uma ação e o estado atual do ambiente escolher qual a próxima ação que deve ser executada.

Durante a implementação diversas abstrações foram criadas no intuito de modularizar e deixar mais claro qual a responsabilidade de cada componente, além de deixar o programa facilmente adaptável em todos os seguimentos. Graças as abstrações podemos mudar o ambiente, o agente, as buscas, os corpos e as mentes sem problemas e sem entrar em conflito em nenhuma parte do programa, pois cada parte está bem estruturada e isolada das demais. Dessa forma temos um programa que possui alta coesão e baixo acoplamento.

Como possibilidade de trabalho futuro podemos incluir um processo de aprendizado ao agente. Qualquer agente pode tirar vantagem do aprendizado, permitindo o mesmo operar em ambientes inicialmente desconhecidos ou então se tornar mais competente que outros tipos de agentes que possuem o mesmo conhecimento inicial. Temos também a possibilidade de implementar na solução do caminho escolhido um processo conhecido como curvas de Bézier, a fim de tornar o caminho definido mais natural, acrescentando-lhe suaves curvaturas. Há ainda há possibilidade da mente utilizar filtros de Kalman, sob certas condições, o filtro de Kalman é capaz de encontrar a melhor estimativa baseada na correção de cada medida de diferentes sensores.

Referências

BELLMAN, R. E. *Otimização Combinatória e Programação Linear*. 1. ed. [S.l.]: Courier Dover Publications, 2003. ISBN 0486428095.

CORMEN, T. H.; RIVEST, C. L. end R. *Introduction to Algorithms*. 2. ed. [S.l.]: McGraw-Hill, 2000. ISBN 0262032937.

DELOURA, M. A. *Game Programming Gems 2*. 1. ed. [S.l.]: Charles River Media, 2000. ISBN 1584500549.

FERNANDES, J. H. C. Ciberespaco: Modelos, tecnologias, aplicacões e perspectivas. In: . [S.l.: s.n.], 2000.

GOLDBARG, M. C.; LUNA, H. P. L. *Otimização Combinatória e Programação Linear*. 2. ed. [S.l.]: Campus / Elsevier, 2005. ISBN 8535215204.

LATTARI, L. G. *Gerenciamento e Visualização de Elementos Virtuais*. 2007.

LAU, M.; KUFFNER, J. J. Behavior planning for character animation. In: *2005 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*. [S.l.: s.n.], 2005. p. 271–280.

LAVALLE, S. M. *Planning Algorithms*. 1. ed. [S.l.]: Cambridge University Press, 2006. ISBN 0521862051.

MONTEIRO, L. H. A. *Sistemas Dinâmicos*. 1. ed. [S.l.]: Editora Livraria da Física, 2002. ISBN 85-88325-08-X.

PATEL, A. *Amit's Thoughts on Pathfinding*. [S.l.]: Stanford University, Internet, 2007.

PRESSMAN, R. S. *Engenharia de Software*. 6. ed. [S.l.]: McGraw-Hill, 2006. ISBN 85-86804-57-6.

RUSSEL, S. J.; NORVING, P. *Inteligência Artificial*. 2. ed. [S.l.]: Editora Campus, 2004. ISBN 85-352-1177-2.

VILLATE, J. E. *Introducão aos sistemas dinâmicos: uma abordagem prática com Maxima*. 1.0. ed. [S.l.]: Creative Commons, 2006. ISBN 972-99396-0-8.

WINK, O.; NIESSEN, W.; VIERGEVER, M. Minimum cost path determination using a simple heuristic function. In: SANFELIU, A. et al. (Ed.). *Image, speech and signal processing*. Los Alamitos: IEEE computer society press, 2000. v. 3, p. 1010–1013.