9,5 (nove e cinco)

Escola Politécnica da Universidade de São Paulo

Departamento de Engenharia Mecatrônica e Sistemas Mecânicos

# Projeto de Formatura

## Simulador de Algoritmos de Otimização de Caminhos para AGVs

1999

Orientador: Prof. Dr. José Sotelo Júnior

Orientado: Vinicius Rodrigues de Moraes

# AGRADECIMENTOS

* Fred, Macel, Cema, Thiago, Sotelo, James, Poli, Valdir, Érica no TF.

* Dodô, Pedro, Jeff, Fred, Marquinhos, Zehh, Menê, Beth, Rogério, Mecatrônica 1992, Érica e pais pela força.

* Professores, Técnicos e Funcionários de toda a USP.

* Gauss, Newton, Mitag-Lafler, Einstein, Blaise Pascal, Euler e Hal Jordan...

* Gary Gygax, Dave Arneson, JRR Tolkien, CARA da Lucas, Raymond Feist, AC Clarke, Stanley Kubrick, David Lynch...

* Massarani, Tsuzuki, Tarcisio e Pedrão Fagundes por serem extrememente gente-boa.

* Net people: Bryan, Steven, Atog, Bjorn, Patel, Dybsand, ...

* Maria Theresa, Jerrizinho e Renata (não acabou ainda!)

# ÍNDICE

# 1. INTRODUÇÃO

O estudo do primeiro semestre de 1999 foi dividido – e realizado em paralelo – em duas partes:

A) estudo da linguagem em que será feito o Simulador, o C++.

B) estudo da teoria para a simulação em si;

Tais estudos foram feitos em pesquisa em livros e na Internet. Relatórios foram passados ao Professor semanalmente, às terças-feiras.

Nas páginas seguintes far-se-á um resumo do que se estudou nas duas partes do estudo.

# 2. LINGUAGEM DE PROGRAMAÇÃO C++

Tipos de Programação:

**1) Não Orientada a objetos:**

1.1) Não-estruturada: cheias de "goto", dificultando demais a programação. Exemplos são o BASIC e o FORTRAN.

1.2) Procedural: seqüências muito utilizadas são agrupadas em procedimentos e funções – tudo ainda no mesmo "main" -, economizando grande trabalho. Não mais se utilizam "desvios incondicionais". Temos o Pascal e o C comum como exemplo. Já se ensaiava um pouco disto com os "gosub" do BASIC.

1.3) Modular: procedimentos e funções de funcionalidade semelhante são agrupados em módulos separados. Um programa deixa, assim, de ter uma única parte. Reutilização de procedimentos e até de módulos inteiros economizam tempo e trabalho; há grande facilitação no "debugging". As linguagens Pascal e C convencional usufruíam da possibilidade de tal programação.

## 2) Orientada a objetos (OO):

2.1) Idéia: é um modo totalmente novo de pensar e programar, até mesmo difícil de enxergar no começo. Mas vamos à frente, ele se mostrará muito poderoso. Aqui, não são as operações e sua hierarquização que definem os dados – como nas linguagens Procedurais -, mas o contrário: os dados ("Abstract Data Types, ADT") criam todo um "habitat" para si, ao seu redor. Alguns conceitos essenciais deste novo modo de pensar:

2.1.1) Objetos: entidades independentes e autônomas, com fronteira conceitual bem definida. Vantagem: é possível isolar uma dada estrutura de dados do resto do programa. Assim, divisão de tarefas entre vários programadores, "debugging", manutenção e melhoramento ficam bastante facilitados.

2.1.2) Classes: são a "argamassa" da linguagem OO. Definem um novo tipo de dado e as operações que se lhes pode aplicar. Por exemplo, uma bola pode ser lançada com as mãos, chutada ou agarrada. Definem-se, também, variáveis e funções. Tudo isto pode ser escondido, se for interessante; isto é algo essencial na programação OO, é o "data hiding". Ficam assim sendo privadas (se não lhes escondermos, são públicas).

2.1.3) Propriedades: são as variáveis citadas acima.

2.1.3) Métodos: são as funções citadas acima.

2.1.4) Nascendo e morrendo: pode-se construir uma variável de inúmeras maneiras; só há uma maneira de a destruir. A alocação de memória é bem mais poderosa que em Pascal ou C.

2.1.5) Herança (mais explicitado abaixo): quando se "herda" de classe(s), está-se criando uma classe baseada na(s) anterior(es), isto é, com os mesmos métodos e propriedades.

2.2) As linguagens OO têm 3 características não presentes nas procedurais:

2.2.1) Encapsulamento de dados ("Data Encapsulation"): é uma bela idéia para proteger uma estrutura que confirmadamente já está "bug-free"; as outras estruturas simplesmente não têm como alterá-la. O contrário vale: altera-se a estrutura sem se mexer no resto do programa.

2.2.2) Polimorfismo ("Polymorphism"): permite a uma entidade ter várias representações. Evitam-se, por exemplo, as chatas mensagens de erro "dupla definição" e suas similares.

2.2.3) Herança ("Inheritance"): nos permite criar tipos de objetos a partir de outro, economizando incrível trabalho. O tipo bola de basquete e o tipo bola de vôlei podem vir do tipo bola (gozam, intuitivamente, de todas as propriedades que seu "pai", não?).

## 2.3) Exemplos de LOO: C++, Delphi e Java.

### 2.3.1) Um bom começo: as linguagens procedurais se adaptam à nova teoria:

"Evoluir para sobreviver": o C ganha novos "poderes" e aparece o C++. O Delphi é a evolução do Pascal, similarmente.

### 2.3.2) Um mundo globalizado, uma nova ordem, uma única linguagem:

Portabilidade, seu nome é JAVA. Já tendo evoluído desde sua criação, é uma ferramenta tão poderosa quanto portável, atualmente, mas ainda lenta por ser "semi-interpretada".

Assim, já se tendo entendido o valor da programação orientada a objetos (grande reaproveitamento, compreensão e projeto de grandes projetos bastante facilitados) pode-se optar por tal método. O C++ apareceu como escolha natural mais específica – dentre as linguagens OO - pela necessidade de sua velocidade superior e pela enorme quantidade de literatura e bibliotecas disponíveis.

A interação gráfica, tanto de entrada como de saída, foi feita em Visual Basic 5, já que não era necessário alto desempenho para esta parte e por ser aquela uma linguagem bastante indicada para se fazer interfaces com usuário.

# 3. INTELIGÊNCIA ARTIFICIAL, BUSCAS EM GRAFOS E OTIMIZAÇÃO

O objetivo da inteligência artificial, segundo Nilsson, é "construir máquinas que realizem tarefas que normalmente requerem inteligência humana".

## 3.1. PROBLEMAS: ESTADOS E OPERADORES

Problema: em vez de definição, usar-se-á exemplo. Ex.: "jogo do 15".

Estado é conjunto definido de variáveis, e não só uma. Ao conjunto possível de estados se dá o nome de Espaço de Estados.

Para discutir métodos de solução de problemas deste tipo é interessante introduzir as noções de estados e operadores. No jogo do 15, cada estado do problema é uma particular configuração dos quadradinhos, como se pode ver na figura 1.

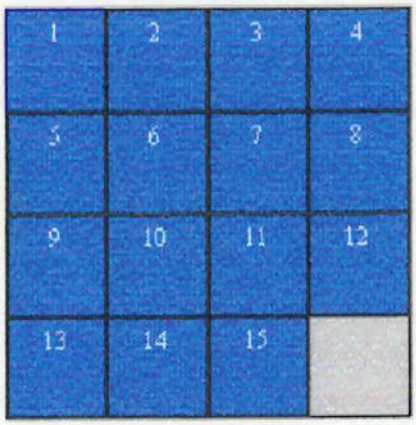



**Figura 1 - Jogo do 15 em um de seus muitos estados (16! no total)**

Tem-se como estados especiais:

* estado inicial:
* estado-objetivo:

O espaço de estados alcançável do estado inicial ao objetivo consiste de todas as configurações legais que podem ser obtidas movendo-se os quadradinhos da maneira permitida pela regra. Este número de estados pode ser grande, muito grande, até mesmo infinito. Ainda no Jogo do 15, este número - para uma dada posição inicial – é de 0,5*16!

O termo "espaço de estados" é emprestado diretamente da Teoria de Controle. A área de Pesquisa Operacional se utiliza também largamente do método de espaço de estados.

Por exemplo: uma aplicação importante dos métodos de espaço de estados é exatamente a solução de problemas de Pesquisa Operacional, sendo o mais

famoso exemplo o do "Caixeiro-viajante", que é um importante modelo para problemas de programação, projeto de produção e logística.

Um operador transforma um estado em outro, exatamente como as Matrizes de Transformação de Estados. No nosso exemplo, tem-se quatro operadores: "mover branco para a direita" (MD), "mover branco para a esquerda" (ME), "mover branco para cima" (MC) e "mover branco para baixo" (MB).

Na linguagem de espaço de estados, uma solução para um dado problema será qualquer seqüência de estados que transformar um estado inicial em um estado-objetivo.

Uma representação interessante: é útil imaginar o espaço de estados alcançável a partir de um dado estado inicial como um grafo. Neste grafo, os nós representam os estados; os arcos orientados que os conectam são os operadores. Isto proporciona uma visão bem mais amigável para a idéia de estados e operadores.

Reservaremos o termo "Método de Espaço de Estados" para os métodos que constróem seqüências de operadores crescentemente (começando, para tal, com algum operador inicial e adicionando um operador por vez até que o objetivo seja alcançado).

## 3.2. REDUZINDO PROBLEMAS A SUBPROBLEMAS

Um método de solução de problemas mais sofisticado envolve a noção de subproblemas.

Cria-se conjunto de subproblemas de modo que a solução de cada um destes implica na solução do problema original.

Por exemplo: o problema "Ir do Brasil para os EUA" pode ser decomposto em "Ir do Brasil para o Panamá" e "Ir do Panamá até os EUA" (sempre via terrestre). É algo semelhante a isto que será utilizado no algoritmo-base de nosso estudo: achar o caminho – no espaço de estados – que minimize o custo de transformação estado inicial/estado-objetivo passando por um dado ponto. Tal problema se reduz a minimizar o custo do início até tal ponto e dele até o fim simultaneamente.

Cada um dos subproblemas pode ser olhado como um novo problema, que pode – bem como o pôde o problema original – ser atacado via espaço de estados ou criação de novos subproblemas.

Pode-se descer tanto no nível de complicação de um problema que se chega em novos problemas que sejam primitivos, isto é, cujas soluções sejam triviais.

Este método descrito neste item é chamado de "Método da Redução do Problema".

Tal método encontra aplicações em sistemas de integração simbólica, por exemplo.

## 3.3. DOIS ELEMENTOS DA SOLUÇÃO DE PROBLEMAS: REPRESENTAÇÃO E BUSCA

Como aprendido em Modelagem e em Controle, deve-se modelar um sistema muito bem antes de se o controlar; a mesma idéia se aplica a nosso estudo: a representação deve ser feita com muito cuidado e critério antes de se iniciar uma busca. Assim, se a faz de maneira rápida.

Qualquer que seja o método escolhido para se solucionar um problema, uma busca pela solução é requerida. Este trabalho é exatamente sobre como se fazer esta busca da maneira mais eficiente possível.

Exatamente por tal importância da eficiência da busca, pode-se acabar – erroneamente – se diminuindo o verdadeiro valor da etapa anterior: a representação do problema a ser resolvido, seja tal representação via espaço de estados, redução de problema ou como um teorema a ser provado.

Concentrando-se na primeira representação, concluir-se-á haver várias delas possíveis para um dado problema, sendo que algumas terão muito menos estados que outras, o que é desejável, sem dúvida, pois representar um problema de

maneira pouco econômica- leia-se com estados em demasia – sem dúvida tornará lento até mesmo o mais eficiente dos algoritmos de busca.

## 3.4. DESCRIÇÕES DE ESTADO

Quando fazendo a formulação via espaço de estados de um problema, é preciso se pensar muito bem em o quê são os estados, e não os confundir com o quê realmente se vai trabalhar no escolhido processo de solução, que são as descrições de estado. No exemplo do jogo do 15, os estados, como já dito, são cada possível configuração das peças. Sua descrição, entretanto, qual é? Uma matriz, uma árvore? Percebe-se, então, ser uma etapa muito importante parte da formulação do problema de espaço de estados é a seleção de alguma particular forma de descrição para os estados de tal problema.

Alguns comentários sobre a escolha de tal forma:

* Qualquer tipo de estrutura de dados pode ser utilizada: matrizes, vetores, listas ligadas, árvore, "heaps", "strings" simbólicos. Muitas vezes sua forma lembra a própria forma do problema em si: no exemplo até aqui utilizado, uma matriz 4x4 aparece logo como uma idéia.

* Outra coisa importante – continuando a idéia de que o cuidado com a preparação do problema é tão importante quanto a sua solução propriamente dita – é utilizar uma forma de representação que facilite as computações dos operadores, acelerando, assim, a transformação de um estado em outro.

Assim sendo, pode-se entender os operadores como funções parciais – não são funções porque nem sempre um operador pode ser aplicado a todos os estados; por exemplo, na figura 1, MD e MB não se aplicam – que, aplicadas no seu domínio – que é o conjunto das descrições de estados possíveis – têm como valor uma descrição. São, assim, do tipo:

$$f: D \rightarrow D$$

onde $D$ é o conjunto das descrições de estados possíveis.

## 3.5. ESTADOS-OBJETIVO

Antes de se tentar chegar ao estado-objetivo, é preciso – novamente preparando a solução do problema - se o entender bem, isto é, se definir completamente a(s) propriedade(s) de tal estado. Só assim será possível se checar se o estado atual é o estado-objetivo, o que indica sucesso da busca.

É bom lembrar, após tal afirmação, que em muitos casos – e o caso deste trabalho é um exemplo – não basta achar uma solução, isto é, um caminho até o objetivo: é necessário achar o melhor caminho. Por "melhor" se entende o caminho que satisfaça um critério de otimização pré-definido, como por exemplo minimizar o número ponderado – via carga computacional - de aplicações dos operadores. Após tal solução, pode-se afirmar que se tem uma solução ótima.

Com o que se viu nesta seção, se pode resumir as etapas de representação de um problema via espaço de estados: três coisas precisam ser especificadas cuidadosamente:

1) a forma da descrição de estados e, em particular, a descrição do estado inicial;

2) o conjunto de operadores e seus efeitos nas descrições de estados;

3) a(s) propriedade(s) da descrição do estado-objetivo.

## 3.6. NOTAÇÃO DE GRAFOS

É bastante interessante expressar o espaço de estados como um grafo orientado. Tal representação é particularmente útil quando discutindo os vários métodos de busca em um espaço de estados.

Introduzir-se-á, portanto, um pouco da terminologia de grafos.

Um grafo é constituído por um conjunto de nós, finito ou não. Alguns dos pares são conectados por arcos (ou bordas), que são direcionados de um dos componentes do par para o outro: daí as denominações arco orientado e grafo

orientado. O grafo de onde "vem" o arco orientado é chamado de pai (ou predecessor) do outro nó, que é chamado de sucessor do primeiro.
Fonte é um nó sem predecessor. Sumidouro é um nó sem sucessor.

No caso de espaço de estados, os nós representam as descrições de estados; os arcos representam os operadores, que transformam um estado em outro.

Uma seqüência de nós como mostrada na figura 2 é chamada trilha* de comprimento k do nó 1 ao nó k. Se existe uma trilha de um nó qualquer i para outro j, j é dito acessível a partir de i, ou descendente de i. O nó i é, assim, ancestral de j. Percebe-se, portanto, ser o problema de achar uma seqüência de operadores que transforme um estado em outro equivalente ao de achar uma trilha num grafo.

Será conveniente, como visto na figura 2, associar custos (pesos) aos arcos. São estes os custos de se aplicar o operador correspondente. Será usada a notação *c(ni,nj)* para denotar o custo do arco que liga o nó i para o nó j, isto é, para se "ir" do primeiro para o segundo, ou ainda para se "levar" o sistema do estado i para o j. O custo de uma trilha qualquer entre dois nós será a soma de todos os arcos que ligam os nós de tal trilha.

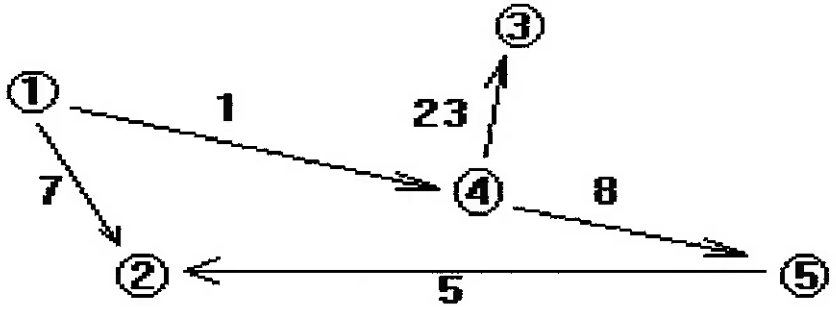



**Figura 2 - Exemplo da estrutura. Grafo orientado (notar os estados - nós - e os operadores - arcos)**

Chama-se usualmente de caminho, mas se chamará o caminho no grafo de trilha para se diferenciar do caminho no mundo real, isto é, o caminho do robô. Pode-se compreender facilmente esta diferença comparando-se as figuras 2 e 3: a primeira é sobre a estrutura de dados, sendo a segunda sobre o mundo real em que estará(ão) o(s) robô(s), alvo(s) de nossa análise.
Com a nomenclatura apresentada, tem-se, na figura 2:

* Nós: 1, 2, 3, 4 e 5.
* Trilhas: 1→2, 1→4→5→2, 1→4→3 e seus subtrilhas.
* Arcos ou Bordas: 1→2, 1→4, 4→5, 5→2 e 4→3.

* Pais ou Predecessores de 1: não há: 1 é fonte.
* Pais ou Predecessores de 2: 1 e 5.
* Sucessores de 4: 3 e 5.
* Sucessores de 2: não há: 2 é sumidouro.

**peso do arco 4→3 é 23. O peso da borda 4→5 é 8.**

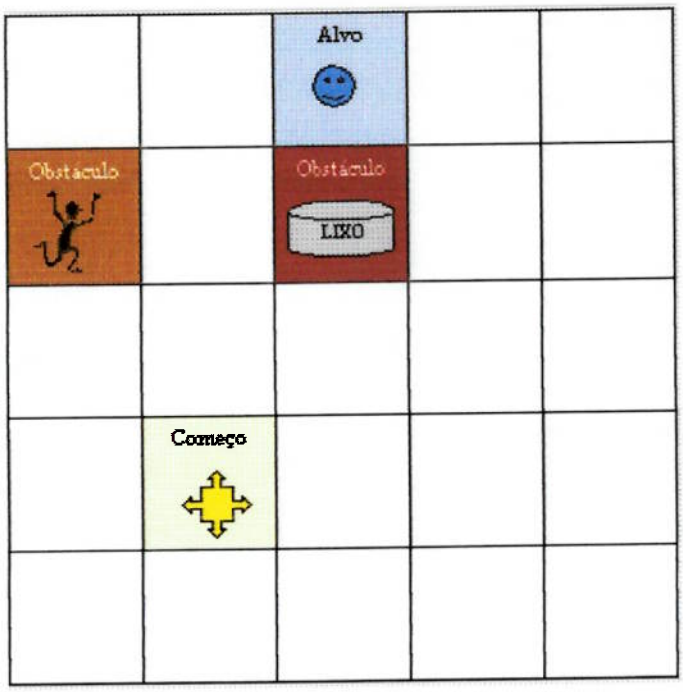



**Figura 3 - Exemplo do ambiente e sua discretização**

Otimização. Com relação a espaço de estados, o que é otimizar? É se tentar achar um conjunto de operadores – que leve do estado inicial até o estado-objetivo - que minimize algum critério pré-definido. No grafo, isto será achar uma trilha que minimize o custo entre os dois nós correspondentes a estes estados, os quais se denominará *s* e *t*, respectivamente.

No caso geral, se quer achar a trilha com custo mínimo entre dois nós quaisquer.

O estado-objetivo pode ser não um, mas sim um conjunto de estados-objetivo. Por exemplo, após receber uma peça bruta, o robô deve levá-la a um torno. Há vários tornos disponíveis. A solução será o caminho mínimo entre todos os possíveis caminhos a todos os tornos. Pode haver, da mesma maneira, um

conjunto de estados iniciais em vez de apenas um: é o caso de vários robôs à disposição para buscar uma peça que acabou de ficar pronta na retífica.
Juntando-se os dois casos, pode-se ter que qualquer um dentre vários robôs pode pegar uma peça na retífica e levá-la a qualquer uma das várias esteiras.

A definição do estado-objetivo (ou dos estados-objetivo) pode ser feita não explicitamente, mas sim implicitamente através das propriedades da descrição de tal estado (ou tais estados), e isto é bastante interessante. Perceber que se insiste novamente aqui na diferença entre um dado estado e sua descrição. No caso da oficina, por exemplo, os estados-objetivo poderiam ter a propriedade suficiente: "faz polimento".

Um grafo pode ser especificado explicitamente ou implicitamente, como se verá.

Numa especificação explícita, os nós, arcos e custos de tais arcos são dados explicitamente. Isto pode ser implementado, por exemplo, através de uma tabela que liste todos os nós do grafo, seus sucessores e os custos associados aos arcos que levam até tais sucessores. Para um número muito grande de arcos, percebe-

se facilmente, uma especificação explícita é impraticável; para um número infinito, é impossível.

Aparece, assim, a especificação implícita, onde um conjunto finito de nós é dado como nós iniciais. Ainda, é dado um operador-sucessor $\gamma$ tal que possa ser aplicado a qualquer nó, dando todos os seus sucessores e os custos dos arcos associados.

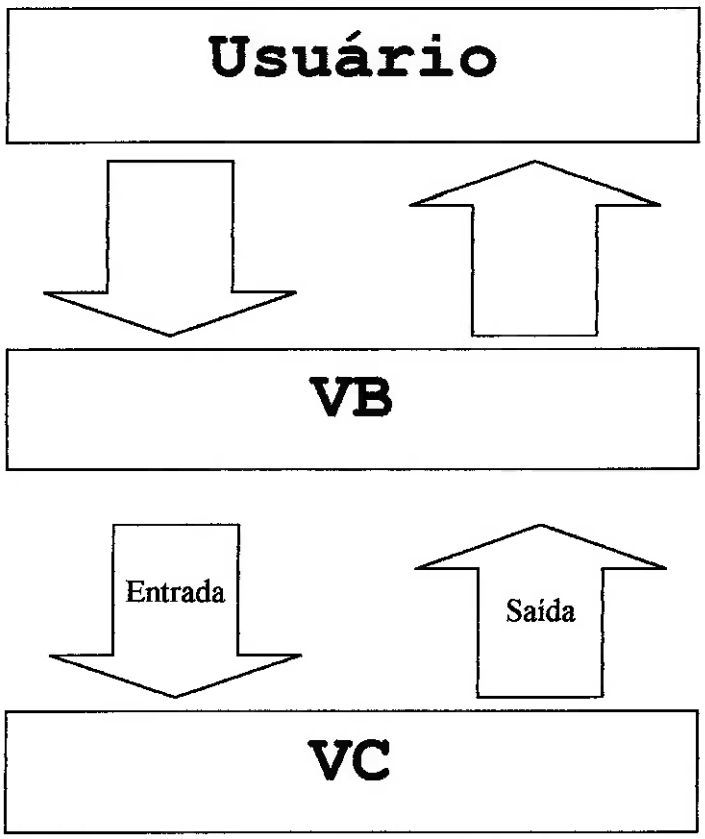
Com sucessivas aplicações de $\gamma$, cobre-se todo o grafo, e isto é feito com uma representação bem mais econômica, isto é, somente em termos do conjunto de estados iniciais e do operador $\gamma$.

O processo de procurar num espaço de estados uma seqüência de operadores que seja solução corresponde a fazer explícita uma porção de um grafo implícito, porção esta suficiente para incluir um nó-objetivo. Procurar em grafos desta maneira é, assim, um elemento central da solução de problemas via espaço de estados.

Como o trabalho é sobre otimização em buscas e se discutiu haver 3 etapas básicas em buscas: representação, estrutura de dados e a busca em si: deve-se otimizar cada uma destas etapas, todas são muito importantes.

# 4. IMPLEMENTAÇÃO

Dado o que se discutiu no texto sobre liguagens de computador, a estrutura do programa Astar.exe (de "A*", o algoritmo utilizado) é:




com a utilização do C já justificada e a do VB também.

No diagrama acima:

* Entrada: ambiente: tamanho do ambiente, pesos e posições inicial e final;
* Saída: nós-solução.

Segue a explicação a respeito do que se implementou até o presente momento. É interessante ressaltar que o programa continua evoluindo. Far-se-á um local na Internet para abrigá-lo, servindo assim à Comunidade Científica. Lá também estará o código aberto.

## 4.1. DESCRIÇÃO DOS ARQUIVOS CONTIDOS NO PACOTE

Arquivos-fonte: *cpp* (C++ "source files") e *h* (C++ "header files"):

* mapa.cpp e mapa.h -> definem a classe CMapa (dimensiona o mapa e confere os pesos).

* posicao.cpp e posicao.h -> definem a classe CPosicao, que indica a posição dentro de um mapa e cuida do deslocamento.

* node.cpp e node.h -> definem um CNo (nó) do processo de determinacao da resposta (um nó tem uma CPosicao, um peso, um passo e um F). Os nós são usados pelo método de "pathfinding" ("Best-First", "Breadth-First", A* etc).

* bestfrst.cpp, bestfrst.h -> todo o A* está aqui. O nome foi mantido por razões históricas, pois foi o Best-First o primeiro método implementado. Perceba que o executável se chama "*astar.exe*".

* astar.cpp e astar.h -> a matriz do mapa está escrita no próprio código. Usa CMapa::Pesos para colocar todos os pesos de uma vez no mapa.

**Listagens:**

Vide arquivos-fonte no CD anexo.

## 4.2. O PROGRAMA

Arquivo executável: "*astar.exe*", compilado com o VB 5.0; astar.dll, com VC++ 5.0.

* Grava e carrega; permite editar um ambiente antigo após o carregar, editar um totalmente novo;
* Entrada do usuário para o programa: visual, ambiente gráfico intuitivo (é a metade da parte feita em VB);
* Formato de entrada para astar.dll (a parte em C++ do programa): matriz em um arquivo-texto, passada pela parte em VB. Tal matriz é uma representação – em formato por nós escolhido – dos dados entrados pelo usuário no ambiente gráfico: tamanho do ambiente, pesos e posições inicial e final. O usuário, como já dito, só precisa interagir em ambiente gráfico, mas pode editar a matriz caso queira. Tais arquivos-texto têm terminação "*hal*", em homenagem ao HAL 9000 e

podem ser modificados em editores de texto comuns. Recomenda-se renomear um arquivo hal antes de se o modificar. Perceba que o arquivo-texto é salvo a cada vez que se calcula o caminho. O formato do arquivo é:

<numero de colunas (dx)>
<numero de linhas (dy)>
<(x=1, y=1),(x=2, y=1),...,(x=dx, y=1),>
<...,>
<(x=1, y=dy),(x=2, y=dy),...,(x=dx, y=dy),>
<posicao start sx,sy,>
<posicao goal gx,gy,>

* Resposta de astar.dll: ele manda para o VB os nós-solução, como coordenadas do ambiente; manda, também, os valores de f, g e h para cada nó. Tais valores serão utilizados na resposta em baixo nível, que pode ser acessada pelo usuário;
* Resposta de astar.exe, isto é, saída do programa para o usuário: no mesmo ambiente gráfico utilizado par a entrada gráfica, é traçado o caminho calculado por astar.dll, que é a resposta desejada. Também se pode ver o resultado em baixo nível, isto é, uma tabela com os valores de interesse da busca: f, g e h para cada nó. Na verdade, o usuário pode estar com ambas as janelas – gráfica e de relatório – abertas ao mesmo tempo. Isto é muito didático e altamente recomendado pelo autor para se entender algoritmo e se ver progresso da busca.

# 5. A METODOLOGIA

## 5.1. UM POUCO DE NOTAÇÃO

Será interessante para a explicação do método de busca empregado – que vem a seguir – introduzir dois termos que serão utilizados com freqüência:

* expandir um nó: é verificar os oito apontadores de dado nó (isto é, aplicar-se os operadores possíveis dentre dos oito existentes: N, NE, E, SE, S, SW, W e NW), obtendo-se assim os filhos daquele nó;

* nó mais promissor que outro: é um nó que obteve uma melhor nota no sistema de medida imposto; este sistema é exatamente a o conjunto de Regras Heurísticas, que é o cerne de boa parte dos algoritmos de Inteligência Artificial e também a parte mais crítica e que requisita um grande tempo de ajuste fino, bem como a escolha de regras de seleção – que vem a ser o "habitat" – em Algoritmos Genéticos. Em ambos os tipos de algoritmo, é este o modo de se comparar as possíveis soluções e se convergir para a resposta do problema.

## 5.2. A DISCRETIZAÇÃO

Quando se vai fazer utilização de algoritmos deste tipo, deve-se escolher entre representação por quadrados ("tiles", ou azulejos) e por "hex" (hexágonos).
Também se deve escolher a respeito da possibilidade de movimento diagonal para o caso quadrado. Perceba que, no caso da discretização em hexágonos, os movimentos possíveis são aqueles nas seis direções intuitivas: para os seis hexágonos adjacentes.

Tais escolhas são ligadas entre si e devem ser feitas na etapa de concepção, assim cedo se decidiu por representação e discretização por azulejos e pela permissão de movimento diagonal.

Preferiu-se não se pôr a listagem em anexo a este texto, pelo fato de se estar fornecendo, junto com ele, o CD com o código aberto. Os arquivos-fonte, fornecidos, podem ser abertos, analisados e modificados pelo leitor, desde que sempre se passe o original a frente. As modificações devem ser submetidas ao autor apra que se organizem as versões, com em qualquer programa de código aberto. O endereço eletrônico é vini@engineer.com.

Melhorias e modificações (novas heurísticas para h, discretização em hexágonos, melhoria da estrutura de dados, novos métodos de busca, novos ambientes em formato *hal* etc) são muito bem-vindas. Sugestões serão debatidas e dúvidas serão respondidas com muito prazer pelo autor no endereço eletrônico dado acima.

Além do código aberto, o CD contém:

* vários arquivos de ambiente (arquivos *hal*);
* programa instalável, em formato *InstallShield.*

## 5.3. ÓTIMO X ADMISSÍVEL: CONFUSÃO COMUM

É muito interessante ressaltar, neste ponto, a diferença entre estes dos importantíssimos conceitos:

* Admissível: é o que normalmente se chama de ótimo, num caso geral. É a garantia de mínimo custo entre dois estados num dado espaço de estados, para dada heurística. Custo mínimo de execução.
* Ótimo: para se achar tal caminho admissível, como dado acima, garante que se expandem o mínimo possível de nós na busca. Custo mínimo de planejamento.

Assim, um dado **caminho** pode ser admissível ou não. E a **busca** para se o achar pode ter sido ótima ou não. Isto faz de um algoritmo de busca:

* admissível ou não;
* ótimo ou não.

# 6. ALGORITMOS CONSAGRADOS

Existem dois tipos básicos de buscas para um problema destes, quanto à sua informação:

* Busca cega, em que não se leva em conta a posição do alvo, tendo-se ou não tal informação; no caso deste trabalho, tem-se tal informação: por que não a usar? Exemplos de algoritmos "cegos":
  - Força Bruta
  - Largura Primeiro
  - Profundidade Primeiro
  - IDDF ("Iterative-deepening depth-first")

* Busca não-cega, em que se utiliza a informação desprezada pelos métodos cegos, obtendo-se rápido direcionamento da busca em direção ao alvo.

Exemplos:
  - Melhor Primeiro
  - Dijkstra
  - A*, que reúne as características positivas dos dois métodos acima,
  - Dijkstra e Melhor Primeiro

# 7. A ESTRELA DOS ALGORITMOS DE BUSCA

## 7.1. COMPARAÇÕES

Como introduzido acima, o A* é, na pior das hipóteses, igual a os dois que o originaram, e aqueles podem ser vistos como casos específicos deste. Assim, já se pode ver vantagens, ainda sem argumentação em mais baixo nível, do A* em relação aos demais métodos de busca.

A argumentação em termos mais técnicos se faz necessária e será feita: dado que o Melhor Primeiro não é admissível – mas muito rápido na busca - e que o Dijkstra é admissível mas não ótimo, por que não unir as características desejáveis de ambos? Nasce o A*.

É interessante salientar que o Melhor Primeiro é muitíssimo interessante quando não se impõe admissibilidade, isto é, quer se achar algum caminho até o alvo com o mínimo tempo de busca possível. Neste caso, o Melhor Primeiro é mais adequado que o A*.

Para o desenvolvimento tecnológico atual - onde sem dúvida o tempo de cálculo de um caminho é ordens de grandeza menor que sua execução, isto é, o que o deslocamento – o A* é bem mais interessante, pois o pequeno tempo perdido em busca com relação ao Melhor Primeiro gera uma enorme economia de tempo, espaço e risco. Para o caso em que o processador fosse muito lento ou que

calculasse caminhos para muitos AGVs ao mesmo tempo, Ter-se-ia um caso em que o custo de busca não mais seria desprezível, e assim o Melhor Primeiro cresceria em importância.

É um enfoque interessante pensar que o Melhor Primeiro só economiza tempo, enquanto o A* minimiza o que quer que seja usado como peso dos azulejos.

O Dijkstra acha o mesmo caminho que o A*, porém com um custo computacional maior. O fato de ser o A* admissível e ótimo quer dizer que nenhum outro algoritmo de busca, utilizando a mesma heurística, acha um caminho admissível expandindo menos nós que o A*.

Assim, a conclusão de interesse é:

* No caso geral, o melhor algoritmo entre os clássicos citados é o A*. Por quê?
* Ele é ótimo e admissível, assim o seu conjunto planejamento/execução é o melhor possível para os problemas a que se propõe resolver; tem mínima arborescência.

## 7.1. FUNCIONAMENTO DO A*

O algoritmo se baseia na equação

$$f^{\wedge}(n) = g(n) + h^{\wedge}(n)$$

onde:

*n* é o nó atual, genérico;

*g (n)* é o custo acumulado desde o nó *S* ("S" de "start"), inicial, até o nó atual, *n*; é um número conhecido a cada instante, e não um estimador;

*h^ (n)* é a estimativa heurística de distância desde o nó atual, *n*, até o nó *G* ("G" de "goal"); o índice " ^ " indica ser um estimador, e não um número conhecido;

*f^ (n)*, sendo a soma de um número com um estimador, é um estimador. Das definições de *g (n)* e de *h^ (n)*, tem-se que *f^ (n)* é o estimador do custo de S a G passando-se pelo nó n. É interessante rever o exemplo da ida do Brasil até os Estados Unidos passando pelo Panamá, mais acima no texto.

Quando se chega ao nó-solução, G, tem-se *h^=0* e, é claro, nada mais a se estimar, então todo o f é composto por g e, assim, o custo é real, número

conhecido. É exatamente este valor que o A* promete minimizar para dados heurística e ambiente.

Por que há tanta insistência no termo heurística? O leitor já deve Ter percebido que é exatamente nela que está a inteligência, a "intuição" do método. É ela que vai "perceber" quais nós são mais promissores. É exatamente nela que reside a diferença entre um bom e um mal A*. Perceba ser ela inexistente em Dijkstra, que só se preocupa com o custo até o nó atual. Perceba também ser a única coisa existente no Melhor Primeiro, que só se preocupa com o custo do trecho ainda faltante até o alvo.

Assim, tem-se que a heurística, no caso do A*, está justamente no $h\hat{}$. Ele é o estimador.

"Heurística" é exatamente uma regra ou conjunto de regras que se usa, vinda de conhecimento e observação anteriores, de bancos de dados. No xadrez, por exemplo, atribuem-se valores às peças e multiplicadores para tais valores, baseados em suas posições a cada momento e no próprio momento do jogo, pois se sabe que um cavalo, por exemplo, é mais valioso no começo do jogo do que no meio ou fim. E também se sabe que uma peça vale mais no meio do tabuleiro do que nos cantos, por causa do número de movimentos disponíveis e de domínio de centro. Perceba: foi dito "sabe-se". A heurística é exatamente isto.

No caso do A*, tem-se que usar um $h\hat{}$ que seja um sub-estimador do valor real, para que não se descartem nós válidos – e, assim, caminhos válidos. No caso de planejamento de trajetórias, utilizam-se um dos seguintes métodos:

* distância Manhattan: dx + dy
* distância Euclidiana: SQRT ($dx^2 + dy^2$)
* atalho diagonal: "anda-se" na diagonal até se estar na mesma linha ou coluna que nó desejado, quando se vai reto até ele, isto é, numa das quatro direções cardeais
* max (dx, dy)

Neste trabalho, foi utilizado o primeiro. O segundo e o terceiro também são bastante interessantes no caso de se permitir movimento diagonal, que é o caso deste trabalho.

Perceba que o estimador não leva em conta o custo do terreno! É exatamente isto que faz o Melhor Primeiro não se preocupar com peso, mas só em ir em direção ao alvo.

Se for usado um sub-estimador muito baixo, a ramificação fica grande, isto é, a heurística ruim começa a "brigar" contra a característica ótima do A*.

Assim, um caminho e a busca por ele são funções do método de busca e da heurística empregados.

Com o colocado acima, já se pode entender o algoritmo A*, apresentado aqui em português estruturado:

(1) Põe o nó inicial s numa lista chamada ABERTOS e calcula f^ (s)

(2) Se ABERTOS está vazia, sai, dando erro em encontrar nó-solução; senão, continua

(3) Remove de ABERTOS o nó com o menor valor de f^ e o põe numa lista chamada FECHADOS. Chama este nó de n. Desempata mínimo f^ arbitrariamente, mas sempre em favor de nó-solução

(4) Se n é nó-solução, sai dando o caminho-solução, obtido rastreando-se de volta os apontadores (postos no passo (5)); senão, continua

(5) Expande o nó n, gerando todos seus sucessores. Se não há sucessores, vai para (2). Para cada sucessor $n_i$, calcula f^ ($n_i$). Põe estes sucessores em ABERTOS, associando-os aos valores recém-calculados de f^, e dá-lhes apontadores de volta para n

(6) Vai para (2)

Assim, o que se faz é : analisam-se, em cada estado do grafo (posição, no mundo real), os nós gerados pelos possíveis operadores. Ordenam-se estes nós-filhos com relação ao seguinte critério: os nós mais "promissores" são postos (numa lista chamada "open", pois estes são os nós ainda abertos a análise) em primeiro lugar, de modo "FIFO" ("first-in-first-out", uma fila). São estes os nós que serão expandidos: seus filhos analisados e assim por diante. Assegura-se, assim, neste método, que os nós mais promissores serão expandidos antes, direcionando-se a busca.

# 8. CONCLUSÃO

O A* é um algoritmo de busca que colabora na resolução de muitos tipos de problemas. Ele resolve:

* problemas de otimização de uma variável num espaço de estados
* problemas de modo admissível e ótimo

O A* não é uma panacéia. Tal coisa não existe. Ele não resolve:

* problemas de otimização de n variáveis simultaneamente (otimizar n-upla, vetor); na verdade, pode-se fazer um "score" para cada estado, baseado na ponderação dos valores de todas as propriedades que se está querendo minimizar; usa-se este "score" como entrada para o A*, já é o caso normal, de só uma variável;
* busca por vários estados (não só um), dada qualquer ordem (caixeiro-viajante generalizado); na verdade, pode-se fazê-lo buscar pelas possibilidades todas de ordem: a análise combinatória irá fazer, para o caso de alguns poucos pontos, o tempo de busca ser muito grande;
* espaço de estados dinâmico; há uma proposição de solução via A* - na verdade o A* Dinâmico, conhecido como D* - por Tony Stentz, CMU.

Parece estar claro que o A* não é o ideal para todos os casos com que se pode deparar na otimização de caminhos, assim se faz necessário uma nova metodologia.

# 9. PROPOSTA: SOLUÇÃO HÍBRIDA

A proposta para a solução de problemas onde o A* não pode ser empregado é a utilização de Solução Híbrida.Tal tipo de solução vem sendo utilizado com sucesso, pois se obtém os pontos positivos de cada uma das metodologia empregadas, sejam elas:

* Lógica Nebulosa;
* Redes Neurais;
* Teoria dos Jogos;
* Sistemas Especialistas;
* Busca Heurística;
* Programação Genética.

E suas características positivas:

* Generalidade;
* Desempenho;
* Aprendizado;
* Solução dinâmica.

Duas ou mais das técnicas acima vêm sendo usadas conjuntamente com bons resultados. A proposta é se unir o trabalho atual – que se utiliza de Busca Heurística – a outra(s) metodologia(s) em um Mestrado neste mesmo Departamento.

# 10.REFERÊNCIAS BIBLIOGRÁFICAS

COLLINS, WILLIAM J., **Programação Estruturada com Estudo de Casos em Pascal.** São Paulo: McGraw-Hill, 1988.

ECKEL, Bruce. **C++: Guia do Usuário.** São Paulo: Makron, McGraw-Hill, Inc., 1991.

LAFORE, Robert, The Waite Group. **Object-oriented Programming in C++ .** 2ª Edição - Waite Group Press, 1995.

MARTINS, Thiago de Castro. **Introdução ao C++ e à Programação Orientada a Objetos.** (Apostila PET). São Paulo, 1997.

NILSSON, Nils J. **Problem-solving Methods in Artificial Intelligence.** McGraw-Hill, Inc., 1971.

NILSSON, Nils J. **Principles of Artificial Intelligence.** McGraw-Hill, Inc., 1980.

TSUZUKI, M. S. G. e outros - **Apostila de PMC-394.**

WIRTH, NIKLAUS. **Algorithms & Data Structures**. New Jersey: Prentice-Hall, Inc., 1986.

INTERNET: Newsgroup comp.ai

INTERNET: Newsgroup comp.ai.games