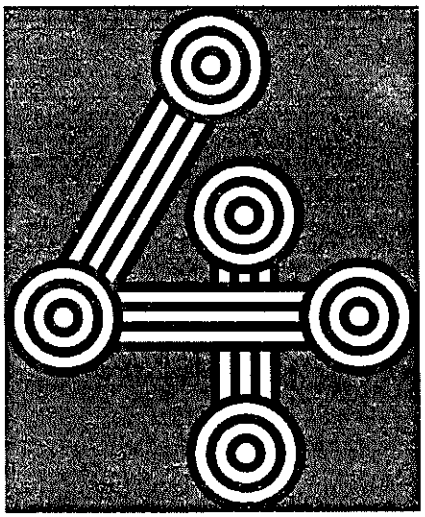

IV SIMPÓSIO BRASILEIRO DE AUTOMAÇÃO INTELIGENTE

08 a 10 de setembro de 1999
Escola Politécnica da USP
São Paulo - SP

Escola Politécnica da USP

ISBN 85-87469-01-0

SBA – Sociedade Brasileira de Automática

8 a 10 de setembro de 1999
São Paulo - SP

Anais do

Quarto Simpósio Brasileiro de Automação Inteligente

Promoção
Sociedade Brasileira de Automática
Escola Politécnica da USP

Editores
Marcelo Godoy Simões – USP
Newton Maruyama – USP

Laboratório de Automação e Sistemas - Mecatrônica

Apoio
FAPESP - Fundação de Amparo à Pesquisa do Estado de São Paulo
CNPq - Conselho Nacional para o Desenvolvimento Científico e Tecnológico
FINEP – Financiadora de Estudos e Projetos

ALGORITMOS DE NAVEGAÇÃO PARA UM ROBÔ MÓVEL UTILIZANDO PROGRAMAÇÃO GENÉTICA

J222a

Daniel Vidal Farfan
Dep. de Engenharia Elétrica
Universidade de São Paulo
São Carlos - SP
dfarfan @ sel.eesc.sc.usp.br

Marco Henrique Terra
Dep. de Engenharia Elétrica
Universidade de São Paulo
São Carlos - SP
terra @ sel.eesc.sc.usp.br

Aluízio Fausto Ribeiro Araújo
Dep. de Engenharia Elétrica
Universidade de São Paulo
São Carlos - SP
aluizioa @ sel.eesc.sc.usp.br

Abstract: The main objective of this paper is to use genetic programming to generate navigation algorithms for a mobile robot that follows walls. The strategy proposed in [2] is modified considering different initial positions for the robot, parameters, and taxes of reproduction, crossover and mutation. Such modifications improve the performance of the robot. A study of parameter robustness is also presented.

palavras chaves: Programação genética, algoritmos genéticos, robô móvel.

1 INTRODUÇÃO

Neste trabalho utilizamos a programação genética para encontrar a solução de um problema particular. Desejamos obter um algoritmo de navegação que conduza um robô móvel a uma distância aproximadamente constante das paredes, por todo o ambiente desconhecido.

Na seção 2 é feita uma breve descrição de programação genética, na seção 3 formula-se o problema descrevendo-se o robô e as salas de treinamento, na seção 4 propõe-se a solução para o problema descrito, e na seção 5 a implementação realizada é apresentada. Finalmente, na seção 6 os resultados obtidos são apresentados.

2 A PROGRAMAÇÃO GENÉTICA

A programação genética, assim com os algoritmos genéticos, é uma técnica de aprendizagem de máquinas que procura imitar os mecanismos de seleção natural na busca da melhor solução para um determinado problema. A principio é gerada uma população inicial de indivíduos utilizando alguma heurística, então se avalia o desempenho de cada indivíduo, e de acordo com seu desempenho, cada indivíduo recebe um número representando o seu desempenho. Esse número é chamado fitness. Os indivíduos de melhor fitness são combinados por meio de operações criativas para gerar a população da próxima geração, então é testado o desempenho desses novos indivíduos e cada indivíduo recebe um fitness. Novas populações vão sendo geradas dessa forma até que o fitness de algum indivíduo ultrapasse um certo limite pré-definido. Esse indivíduo é a solução do problema em questão.

Cada indivíduo da população é um programa de computador sintaticamente válido que possui: funções, procedimentos, e variáveis. Chamamos de funções as operações que recebem argumentos, e terminais as operações que não recebem. Representamos cada indivíduo da população por meio de uma árvore formada por um nó raiz, nós intermediários, e nós folhas. O nó raiz é o que origina a árvore, e fica localizado acima dos demais, os nós intermediários fazem ligação entre outros nós, e os nós folhas são aqueles que não se ramificam. Os nós contém círculos com letras, que representam funções, e quadrados com números, que representam terminais (fig.1).

As operações criativas mais comuns, utilizadas para gerar novas populações, são: reprodução, cruzamento, e mutação.

Reprodução: É a cópia de um indivíduo da geração anterior para a próxima geração, sem qualquer alteração em sua estrutura (Fig.1).

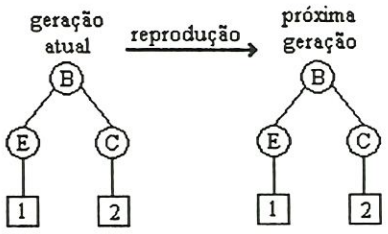

Fig.1: Exemplo de reprodução.

Cruzamento: Consiste em selecionar dois indivíduos, e um nó ao acaso em cada um deles, então trocar os nós selecionados, juntamente com qualquer sub-árvore que exista abaixo deles (Fig.2).

1042870
300799

SYSNO 1042870
PROD 002645
ACERVO EESC

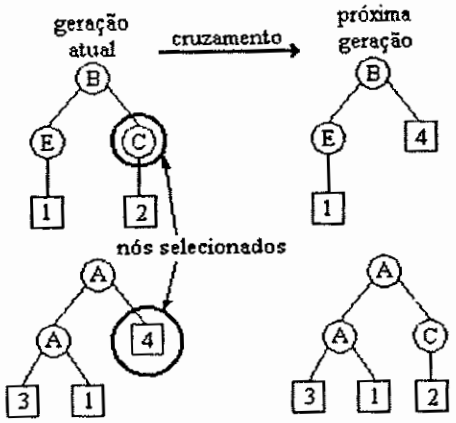

Fig.2: Exemplo de cruzamento.

Mutação: Consiste em selecionar um indivíduo, e um nó ao acaso em sua estrutura, remover o nó, juntamente com qualquer sub-árvore que exista abaixo dele, gerar uma nova sub-árvore e colocar no lugar da antiga (Fig.3).

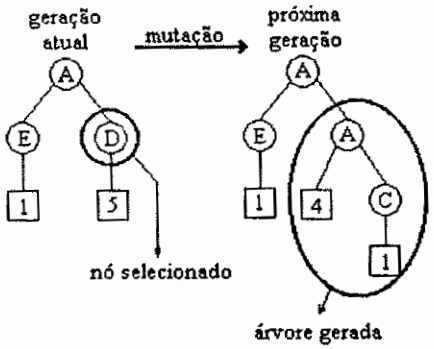

Fig.3: Exemplo de mutação.

Como parte do projeto, é necessário especificar um conjunto de funções e terminais que devem satisfazer as seguintes restrições para que a programação genética consiga encontrar a solução do problema:

Suficiência: Requer que exista uma solução para o problema no domínio de todos os programas que podem ser gerados com o conjunto de terminais e funções especificado.

Fechamento: Requer que toda função e terminal devolva um valor de tipo compatível, e que toda função aceite argumentos desse mesmo tipo.

A suficiência garante que a solução para o problema existe, e o fechamento garante que qualquer árvore completa (todos os nós folhas são um terminal) é um programa sintaticamente válido que pode ser compilado e executado.

Também é necessário definir um teste de fitness que irá avaliar o desempenho de cada indivíduo.

3 DEFINIÇÃO DO PROBLEMA

3.1 O robô seguidor de paredes

O robô simulado neste trabalho pode se mover pelos espaços vazios de seu ambiente, medir a sua distância até as paredes, e mudar a sua direção. O robô possuí oito sensores de alcance posicionados a incrementos de 45° relativos a sua frente (Fig.4): S1, S2, S3, S4, S5, S6, S7, S8. O sensor S1 sempre indica a distância do robô até a parede à sua frente, o sensor S7 até a parede à direita, o sensor S3, até a parede à esquerda, e similarmente ocorrendo com o restante dos sensores.

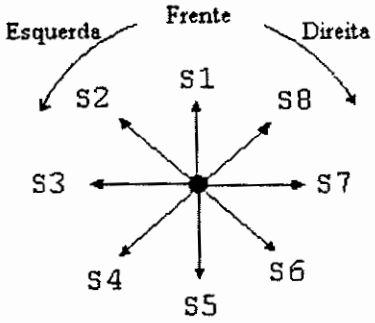

Fig.4: O robô e seus sensores

3.2 As salas de treinamento

Um ambiente bidimensional é simulado, consistindo em quatro salas completamente fechadas, com paredes variando em complexidade (Fig.5). Cada sala do ambiente simulado consiste em uma grade de 40 por 40 células, onde cada célula é um quadrado de lado L. L é um valor que pode variar, e é medido em pixels. As áreas negras representam as paredes do ambiente, as áreas cinzas representam o caminho que o robô deve percorrer, e o símbolo * em cada sala indica a posição inicial do robô naquela sala. Para poder avaliar o desempenho de cada indivíduo, dividimos as áreas cinzas em unidades que denominamos células de corredor. O robô ideal deve visitar pelo menos um pixel de todas as 930 células de corredor existentes.

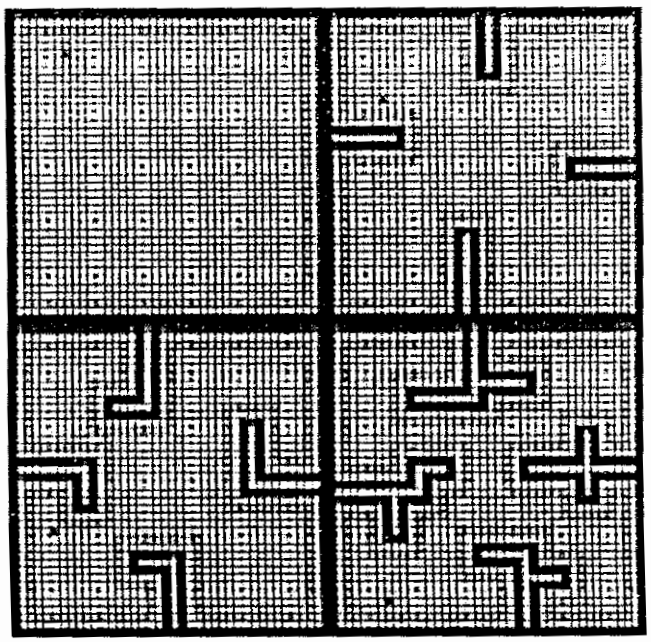

Fig.5: Salas de treinamento

Utilizamos um ambiente com quatro salas para avaliar a importância da posição inicial do robô, e encorajar o desenvolvimento de soluções gerais, evitando assim a simples memorização de um conjunto de movimentos.

O tamanho do pixel é usado como a unidade básica de movimento, e dentro de cada sala o robô pode se mover em 8 sentidos fixos (Fig.6): 1, 2, 3, 4, 5, 6, 7, 8. Os vetores da figura 6 são utilizados para localizar a frente do robô em relação à sala.

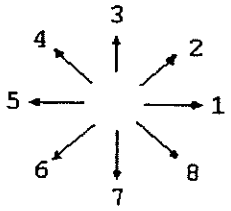

Fig.6: Vetores de orientação

4 PROPOSIÇÃO DA SOLUÇÃO

Nesta seção definimos o conjunto de funções e terminais que será utilizado para resolver o problema, assim como o teste de fitness.

As funções e terminais são definidos com base em três variáveis lógicas que indicam o estado atual do robô: TooCloseToWall (vale 1 se o robô está muito perto da parede), IncoridorRange (vale 1 se o robô está em uma célula de corredor), e TooFarFromWall (vale 1 se o robô está muito longe da parede). Estas variáveis, assim como seus parâmetros a1, d, e δ, são descritas na seção 5.

4.1 Funções

As funções aqui definidas seguem o seguinte modelo: *NomeDaFunção(lista de argumentos que utiliza).*

Do2(Arga, Argb). Esta função faz operações seqüenciais. Ela avalia primeiro o argumento *Arga*, e em seguida o argumento *Argb*. O valor retornado é o resultado do argumento *Argb*.

WhileInCoridorRange(Arga). Enquanto *InCoridorRange* = 1, o argumento Arga é avaliado continuamente. O valor retornado é o resultado da última avaliação do argumento *Arga*.

WhileTooCloseToWall(Arga). Enquanto *TooCloseToWall* = 1, o argumento *Arga* é avaliado continuamente. O valor retornado é o resultado da última avaliação do argumento *Arga*.

WhileTooFarFromWall(Arga). Enquanto *TooFarFromWall* = 1, o argumento Arga é avaliado continuamente. O valor retornado é o resultado da última avaliação do argumento *Arga*.

IfConvexCorner(Arga, Argb). Se a distância medida pelo sensor S3 for maior que (a1+d) e a distância medida pelo sensor S4 for menor que (a1+d) esta função avalia o argumento *Arga*, caso contrário, ela avalia o argumento *Argb*. Em ambos os casos o valor retornado é o resultado da avaliação do argumento.

4.2 Terminais

Os terminais aqui definidos seguem o seguinte modelo: *NomeDoTerminal().* Os terminais não recebem argumentos.

MoveForward(). Este terminal faz o robô avançar 'passo' pixels a sua frente. O valor retornado é 1 se o robô completa seu movimento sem colidir com uma parede, e 0 se uma colisão acontece. Se o robô colide com uma parede, o movimento pára naquele ponto evitando que o robô invada a parede.

TurnRight(), TurnLeft(). Estes terminais fazem o robô girar 45° à direita ou à esquerda, respectivamente. O valor retornado é a nova direção do robô.

TurnTowardsClosestWall(). Este terminal verifica as distâncias medidas pelos sensores e gira o robô em direção à parede mais próxima. O valor retornado é a nova direção do robô.

TurnAwayFromClosestWall(). Este terminal é semelhante a *TurnTowardsClosestWall()*, exceto que o robô gira para a direção oposta à parede mais próxima.

TurnParallelToClosestWall(). Este terminal também é semelhante a *TurnTowardsClosestWall()*, exceto que o robô gira de tal forma que seu sensor S3 é apontado em direção à parede mais próxima.

4.3 Função de fitness

A função de fitness será utilizada para medir o desempenho dos indivíduos, e classificá-los. Ela leva em consideração três parâmetros: *NumberOfCorridorGridsNotVisited, DistanceOfEndingPointFromCenterOfCorridor,* e *TotalDistanceTravelled*

NumberOfCorridorGridsNotVisited: É o número de células de corredor não visitadas.

TotalDistanceTravelled: É o número de células que o robô visitou. É incluído para favorecer os algoritmos que cumprem a sua tarefa caminhando apenas o necessário.

DistanceOfEndingPointFromCenterOfCorridor: É a distância do ponto final do robô até o centro da célula de corredor mais próxima. Este termo é usado para evitar a convergência prematura nas primeiras gerações.

Escolhemos a seguinte função para avaliar o desempenho dos indivíduos da população:

$$10^6 / (1000 * \mathit{NumberOfCorridorGridsNotVisited} + 100* \mathit{DistanceOfEndingPointFromCenterOfCorridor} + \mathit{TotalDistanceTravelled})$$

5 IMPLEMENTAÇÃO

5.1 As variáveis de estado

As variáveis de estado IncoridorRange, TooFarFromWall e TooCloseToWall dependem de três parâmetros pré-definidos: a1 = distância da célula de corredor até a parede, d = largura da célula de corredor, e δ = faixa de tolerancia (Fig.7).

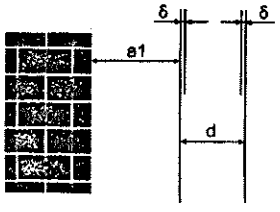

Fig.7: Parâmetros para as variáveis de estado

Sempre que o robô verifica a leitura de seus sensores de alcance, o algoritmo atualizar é executado para atualizar as variáveis de estado (tabela 1).

Tabela 1: Algoritmo atualizar

1-	Atribua a x a distância do robô até a parede mais próxima;
2-	Se ($x < a1 + \delta$) *TooCloseToWall* := 1, caso contrário *TooCloseToWall* := 0;
3-	Se ($x > a1 + d - \delta$) *TooFarFromWall* := 1, caso contrário *TooFarFromWall* := 0;
4-	Se ($a1 \leq x \leq a1 + d$) *IncoridorRange* := 1, caso contrário *IncoridorRange* := 0.

5.2 Funções e terminais

Inspirados no trabalho de Langdon [3], optamos por representar os indivíduos (programas) por meio de cadeias de caracteres (*strings*). A cada função foi atribuída uma letra, e a cada terminal foi atribuído um número, conforme mostrado na tabela 2. A figura 8 mostra os elementos utilizados na construção dos programas.

Tabela 2: Representação

'A'	→ *Do2(argA,argB)*
'B'	→ *IfConvexCorner(argA,argB)*
'C'	→ *WhileInCoridorRange(argA)*
'D'	→ *WhileTooCloseToWall(argA)*
'E'	→ *WhileTooFarFromWall(argA)*
'1'	→ *MoveForward()*
'2'	→ *TurnRight()*
'3'	→ *TurnLeft()*
'4'	→ *TurnTowardsClosestWall()*
'5'	→ *TurnAwayFromClosestWall()*
'6'	→ *TurnParallelToClosestWall()*

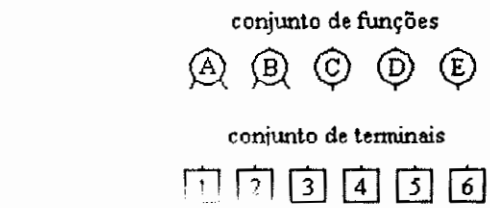

Fig.8: Elementos para a construção das árvores

Utilizando-se esta representação, o indivíduo 2 da figura 3 é representado pela seguinte cadeia de carácteres: "AE1A4C1", e o programa correspondente é o seguinte:

```
Do2
  WhileTooFarFromWall
        MoveForward
  Do2
        TurnTowardsClosestWall
        WhileInCoridorRange
              MoveForward
```

5.3 Considerações gerais

Para obter os resultados apresentados neste trabalho, desenvolvemos um simulador [10], tomando por base [3], escrito em ANSI C, com interface gráfica para o sistema operacional DOS. As figuras 5, 9 e 10 foram geradas pelo simulador, e importadas para este texto. O simulador utiliza um gerador de números aleatórios portável [5], implementado por nós, que irá fornecer os mesmos resultados em qualquer máquina. Utilizamos a roleta como mecanismo de seleção [4]. Todas as experiências foram executadas em um PC Pentium-300 Mhz rodando Windows 95.

A população foi fixada em 200 indivíduos, e o número máximo de gerações utilizado foi 400. Limitamos o tamanho máximo de cada indivíduo a 70 caracteres, e optamos por sempre copiar o indivíduo de melhor desempenho para a próxima geração, impedindo que ele se perca em virtude da aleatoriedade do processo. Estes valores foram fixados levando-se em conta os resultados obtidos em [2].

Durante os testes preliminares identificamos duas formas de evoluir os indivíduos para alcançar a solução desejada. A primeira é gerar uma população inicial com indivíduos grandes (próximos ao limite máximo permitido) e ir combinando-os, principalmente por meio de *cruzamento*. A segunda é gerar uma população inicial pequena e ir evoluindo os indivíduos principalmente por meio de mutação. Neste trabalho utilizamos a segunda forma, devido a ela ter nos fornecido melhores resultados. As probabilidades para reprodução, cruzamento e mutação determinam qual forma será utilizada para evoluir os indivíduos. Neste trabalhos, utilizamos: 30%, 60% e 10%, respectivamente. Estes valores foram selecionados com base no trabalho anterior [2].

É oportuno salientar que os programas gerados freqüentemente entram em loop infinito, e é necessário que na implementação das funções e terminais sejam tomadas providências para evitar esses inconvenientes. Os seguintes procedimentos foram adotados para contornar esse problema: 1° limitamos o número total de passos que o robô pode dar em cada sala a 330xL; 2° sempre que um "loop while" é concluído e o robô não se moveu, o loop é terminado. Esses procedimentos fizeram com que a execução de cada programa terminasse em um tempo finito. O valor 330xL foi escolhido de forma a permitir que o robô caminhe o suficiente para poder visitar todas as células de corredor da sala, antes de "ficar sem combustível".

6 RESULTADOS

Em [2], o autor fixou os parâmetros do simulador nos seguintes valores: L = d = a1 = 10, δ = 2, e passo = 5. Neste trabalho fizemos um estudo paramétrico para verificar a influência desses parâmetros na aprendizagem do robô. Optamos por manter a relação L = d = a1, e variar os valores de L, δ, e passo. Utilizamos dois critérios de parada para finalizar o processo de evolução:

1°- algum indivíduo consegue visitar mais de 90% das células de corredor existentes, ou seja, visitar mais de 837 células das 930 existentes;

2°- o número de gerações chega a 400.

Tabela 3: Estudo paramétrico

L	δ	P	CP	*Fitness*	N1	N2
10	2	1	1°	12.5775	852	277
10	2	2	1°	17.3677	875	922
10	2	3	1°	10.7670	840	1684
10	2	4	2°	6.8750	787	1427
10	2	5	1°	10.9543	840	172
10	2	6	2°	10.0640	833	830
10	2	7	2°	1.1872	89	678
10	4	1	1°	11.1110	843	882
10	4	2	1°	11.1808	842	219
10	4	3	1°	14.1409	861	265
10	4	4	1°	15.9248	869	265
10	4	5	1°	10.9543	840	172
10	4	6	1°	19.0480	879	277
10	4	7	1°	19.6692	881	311
10	4	8	1°	13.0727	855	289
5	1	1	1°	16.5183	871	206
5	2	1	1°	11.0557	841	218
5	1	2	2°	3.2107	620	300
5	2	2	2°	4.5793	714	1325
5	1	3	1°	12.2359	850	279
5	2	3	1°	11.3025	843	368

A tabela 3 apresenta os resultados obtidos, onde tem-se que:

P: Passo utilizado no terminal *MoveForward()*;
CP: Critério de parada que foi acionado;
Fitness: Resultado da função de desempenho;
N1: Número de células de corredor visitadas;
N2: Número de células que visitou, mas que não deveria ter visitado. O número máximo é 3361.

Nesse estudo paramétrico fizemos o robô começar sempre do mesmo lugar em cada uma das quatro salas (veja Fig. 5). Na sala 1 o robô foi colocado no sentido do vetor 3 (Fig. 6), nas salas 2 e 3 o robô foi colocado no sentido do vetor 1, e na sala 4 o robô foi colocado no sentido do vetor 2. Estas posições e sentidos adotados diferem das adotadas em [2], onde Dain optou por colocar o robô sempre no centro de cada sala, e no sentido do vetor 1. Neste trabalho realizamos alguns estudos utilizando as mesmas configurações iniciais utilizadas em [2], e constatamos que essa configuração é muito particular devido a dois motivos: 1° o sentido adotado coloca o robô de frente para a parede mais próxima; 2° a posição é muito simétrica, não ficando muito evidente para o robô qual é a parede mais próxima, caso ele seja colocado em outro sentido. Devido a essas considerações, os robôs treinados na configuração [2] aprendem a andar para frente até encontrar uma célula de corredor, e então caminham acompanhando as paredes. Com as alterações feitas neste trabalho colocamos o robô em posições e sentidos que aumentam o gradiente das distâncias medidas pelo robô, e favoreçem os algoritmos que conduzem o robô para a parede mais próxima, independente de qual seja a posição e sentidos iniciais.

A tabela 3 mostra apenas parte do estudo paramétrico que realizamos, e foi incluída para mostrar ao leitor a influência dos parâmetros na aprendizagem do robô.

Quando utilizamos os parâmetros: L=10, a1=10, d=5, δ=2, P=1, obtivemos o programa "A2EACAA2AEAC63A1C361", denominado robô 1, apresentado na figura 9.

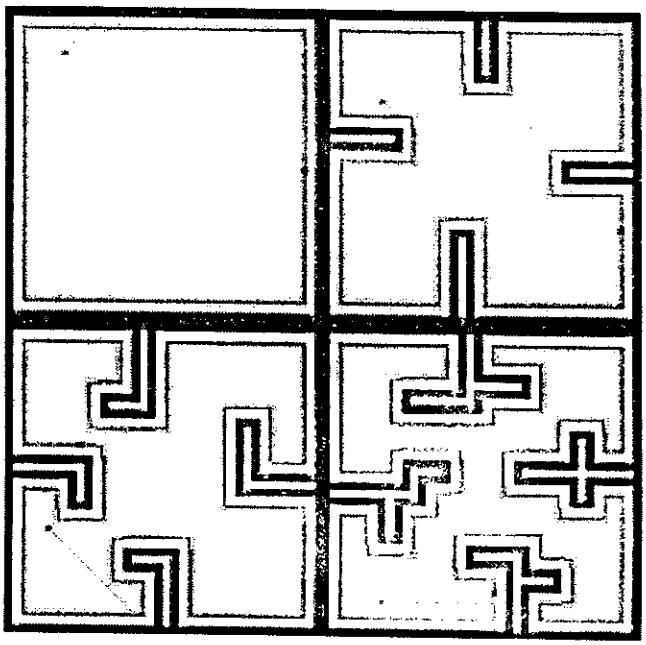

Fig. 9: Comportamento do robô 1

A figura 9 mostra claramente, que o robô 1 apresenta o comportamento desejado, e é uma solução para o problema proposto, pois visita todas as células de corredor do seu ambiente.

Pelo comportamento do robô 1 nas salas 3 e 4 (fig.9), vemos que ele não é a solução ótima para o problema, pois a partir de sua posição inicial, ele não percorre o menor caminho em direção às células de corredor. Na tentativa de encontrar a solução ótima, mudamos apenas o sentido da posição inicial do robô 1 nas salas. O robô foi posicionado no sentido do vetor 1 em todas as salas, e assim obtivemos o programa "EAEA4AAEA4AC61AC214AC21", denominado robô 2, cujo comportamento, quando colocado em diferentes posições inciais e sentidos, é apresentado na figura 10.

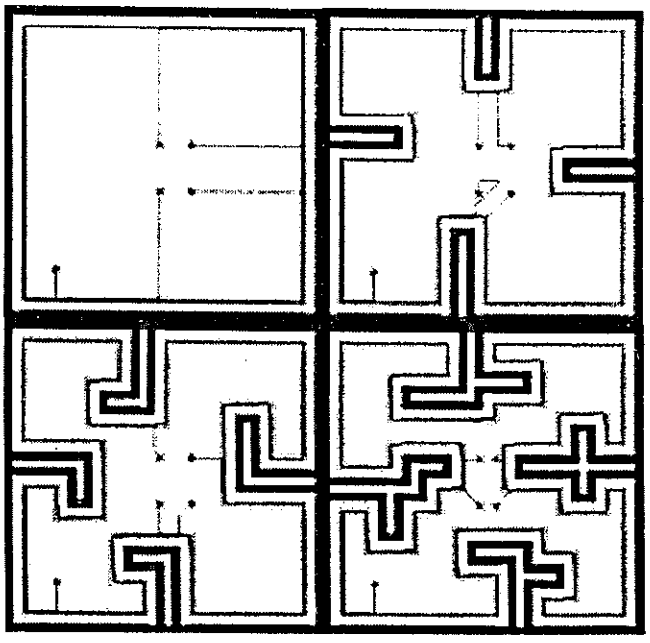

Fig.10: Comportamento do robô 2

7 CONCLUSÕES

Utilizando a programação genética foi possível encontrar uma solução para o problema proposto. Os robôs 1 e 2, são soluções robustas para o problema proposto, pois funcionam em salas diferentes das utilizadas no treinamento. O algoritmo apresentado em [2] não consegue visitar as células de corredor que se encontram nos cantos convexos, ocasionalmente cometem erros, abandonando os corredores, e não se dirigem para a célula de corredor mais próxima. Por essas razões as duas soluções aqui apresentadas tem melhor desempenho quando comparadas com o algoritmo apresentado em [2].

A maneira como se posiciona inicialmente o robô em cada sala é de fundamental importância para se encontrar a solução ótima do problema. Pequenas variações nos parâmetros ocasionam a convergência para algoritmos distintos. Encontrar a melhor solução para o problema exige muitas combinações dos parâmetros, perspicácia do programador, e um pouco de sorte.

Em geral a programação genética nos fornece algoritmos de navegação com relativa facilidade, mas a questão mais difícil é encontrar uma solução que preencha todos os requisitos desejados. De acordo com os parâmetros utilizados, obtêm-se algoritmos que utilizam estratégias variadas para resolver o problema, algumas vezes encontrando estratégias nunca antes pensadas.

8 AGRADECIMENTOS

Agradecemos a CAPES pelo fomento a pesquisa, ao José Geraldo V. Moreira pela programação do módulo de entrada, ao Sr. JESUS pela grande contribuição no módulo de interpretação das strings, e pelo apoio a este trabalho, a T. Patricia Simões, e ao Igor Prata pelas suas contribuições a este trabalho.

9 REFERÊNCIAS BIBLIOGRÁFICAS

[1] J.R. Koza, *Genetic programming*, The MIT Press, 1992.

[2] R.A. Dain, "Developing Mobile Robot Wall-Following Algorithms Using Genetic Programming", *Applied Inteligence*, vol. 8, n.1, January, 1998, pp. 33-41.

[3] W. B. Langdon, *Quick Intro to simple-gp.c*, University College London, 1994.
ftp://cs.ucl.ac.uk/genetic/gp-code/simple/*.*

[4] D.E. Goldberg, Genetic algorithms in search, optimization, and machine learning, Addison-Wesley Publishing Company, 1989.

[5] W.H. Press, et al., Numerical Recipes in C : The Art of Scientific Computing; Cambridge University Press, 1992

[6] K.A. Barclay, ANSI C: problem solving and programming, Prentice Hall Europe, 1990.

[7] P.H. Winston, Artificial Intelligence, Addison-Wesley, 1992.

[8] P.J. Angeline(Editor), K.E. Kinnear (Editor), *Advances in Genetic Programming (Complex Adaptive Systems Series, Vol 2*, MIT Press, 1996

[9] W.B. Langdon, *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, Kluwer Academic Pub, 1998.

[10] ftp://ftp.aml.net/pub/robo/