Um Robô Aspirador em Assembly MIPS
Implementação e Comentários

Kevin Scaccia, Rosângela Farias, William Firmino

[1]ICT – Universidade Federal de São Paulo (UNIFESP)

Abstract. *This paper describes step by step the software implementation process of a Vacuum Cleaner Robot commonly used in homes. The software simulates the robot behavior in a two-dimensional environment where all points need to be cleaned except those marked as points inaccessible to the robot (walls, furniture, and other objects). The software was written in the MIPS Assembly language and debugged in the MARS emulator.*

Resumo. *Este trabalho descreve passo a passo o processo de implementação do software de um robô aspirador de pó comumente utilizado em residencias. O software simula o comportamento do robô em um ambiente bidimensional onde todos os pontos precisam ser limpos, exceto aqueles marcados como pontos inacessíveis ao robô(paredes, móveis e outros objetos). O software foi escrito em linguagem de montagem Assembly MIPS e depurado no emulador MARS.*

1. Descrição do Ambiente

o mapa de bits é a reprodução gráfica (mapa) das variações possíveis nas unidades de memória do computador (bits). O bit é a unidade mínima da memória do computador. É ele quem define o comportamento binário do ambiente digital através de dois valores possíveis: um (ativo) e zero (desativo)

2. Comportamento do Robô

O robô inicia em uma posição aleatória do ambiente bidimensional. A partir dessa posição inicial o robô percorre todo o mapa a procura de espaços que ainda não foram percorridos e para quando estiver certo de que todo o mapa foi explorado. Apesar do fato de que uma busca aleatória pelo ambiente poder ser implementado e o objetivo sempre ser atingido em algum tempo finito, escolhemos por implementar uma **heurística de busca gulosa**, onde o robô sempre escolhe visitar espaços onde ele menos esteve, na esperança de que estes sejam próximos de espaços onde ele nunca esteve. É possível demonstrar que esta heurística reduz o espaço de busca(e consequentemente o tempo de execução) por um fator exponencial em relação à busca totalmente aleatória, isso à medida que o tamanho do problema cresce(dimensão do mapa).

O robo tem o modo de funcionamento aleatório.A partir da posição inicial e definição do número de móveis fixos, o robot avança à procura de lixo para recolher. Ele tem de percorrer todas as posições pintando-as, indicando que já passou por aquela posição e incrementa uma variável inteira que conta o número posições percorridas até percorrer todo a área, desviando-se dos obstáculos.

3. Implementação

Esta sessão explica o desenvolvimento e funcionamento do código implementado, dando maior ênfase ás funções mais críticas do software. Observe que os **comentários mostrados nas listagens** abaixo são iniciados por **;**, isso se deve **somente a sintaxe** do $codehighlight$ para Latex, **os comentários em Assembly MIPS** são iniciados por **#**.

3.1. Segmento de Dados

O segmento de dados possui somente duas cadeias de caracteres correspondentes as mensagens iniciais que são exibidas na interação do usuário, onde ele entra com a quantidade de ojetos(móveis) que serão instanciados aleatoriamente no mapa. O robô então considera essas posições como posições que não podem ser percorridas. Abaixo segue o código:

```
.data
.msgMoveisFixos: .asciiz "insira o numero de moveis que nao ..."
.msgMoveisLivres: .asciiz "insira o numero de moveis que podem ... "
```

3.2. Segmento de Texto

O segmento de texto está organizado da seguinte forma: Inicialmente temos um conjunto de instruções jal(jump and link) que funcionam como chamadas de procedimentos que realizam as definições e configurações iniciais do programa. Cada procedimento é explicado nas subsessões que seguem. Abaixo segue o código que chama cada procedimento.

```
.text
        jal defCores
        jal defMap
        jal desenhaPlanta
        jal nMoveis
        jal insereMovelF
        jal insereRobo
        ; abaixo o codigo referente ao Main Loop
        ; ...
        ; ...
```

3.2.1. Main Loop

Chamamos de Main Loop o trecho do código que se repete enquanto o robô procura por espaços não percorridos no mapa, ou seja, cada iteração de $main_loop$ corresponde a verificação se todo o espaço já foi percorrido. Em caso positivo o robô para e chamamos uma $syscall$ de saída do programa, em caso negativo o robô continua sua busca por espaços não visitados chamando o procedimento $moveRobo$. Abaixo segue o código que implementa essa funcionalidade.

```
addi $t8, $zero, 183 ; quantidade de locais disponiveis
sub $t8, $t8, $s7 ; t8 agora possui o total de posicoes disponiveis para limpeza
main_loop:
        jal moveRobo ; realiza um movimento
        bne $t8, 0, main_loop ; verifica se o robo acabou seu trabalho
```

3.3. Procedimentos de Inicialização

Esta sessão esmiuça cada um dos procedimentos de definição e inicialização necessários para o desenvolvimento do programa.

3.3.1. defCores - Definiçao de Cores do Display

Este procedimento simplesmente atribui a certos registradores valores em hexadecimal de cores usadas no display do programa. Estes registradores são usados durante todo o resto do programa e não podem ser reescritos.

```
defCores:
        addi $s0, $zero, 0x00FF0000 ; registrador s0 recebe a cor vermelha
        addi $s1, $zero, 0x007B68EE ; registrador s1 recebe a cor roxa
        addi $s3, $zero, 0x00FFFF00 ; registrador s3 recebe a cor azul FFFF00
        jr $ra ; retorna ao endereço armazenado no registrador de endereços
```

3.3.2. defMap - Definição do Tamanho do Mapa

Este procedimento define o endereço em memória que representa o ponto incial do mapa (superior esquerdo), no caso o endereço $0x1004$.

```
defMap:
        addi $t0, $zero, 256 ; 100000
        add $t1, $t0, $zero
        lui $t1, 0x1004 ; 0x1004 é colocado como bit mais significativo
        ;t2 como menos significativo
        jr $ra ; retorna ao endereço armazenado no registrador de endereços
```

3.3.3. desenhaPlanta - Cria e Imprime o Ambiente

Este procedimento, o mais denso que os demais, é responsável pelo desenho de um mapa estático, de dimensões NxN fechado por obstáculos em suas extremidades(paredes do ambiente).

```
desenhaPlanta:
        add $t0, $zero, $t1 ; t0 recebe a posição inicial atribuida em t1
        addi $t2, $zero, 0 ; t2 recebe 0 para ser nosso contador
        loop_1: ;Loop de linha vermelha no canto superior esquerdo
                sw $s0, ($t0) ; Pinto o pixel na posicao $t0 com a cor de $s4
                addi $t0, $t0, 4 ; Pulo +4 no pixel, pula 4 bytes
                addi $t2, $t2, 1 ; incrementa contador
                ; se chegar até o final vai para a proxima linha
                beq $t2, 15, exit_1
                j loop_1
        exit_1:

        #t0 foi incrimentado antes de sair do laço
        addi $t2, $zero, 0 ;t2 recebe 0
        loop_2: ; Loop de linha vermelha no canto direito
                sw $s0, ($t0) ; Pinto o pixel na posicao $t0 com a cor de $s4
                addi $t0, $t0, 64 ; Pulo +64 no pixel, pula 4 bytes * 16 casas,
                addi $t2, $t2, 1 ; incrementa contador
                beq $t2, 16, exit_2 ; pinta todos os 16 quadros da linha
                j loop_2
        exit_2:

        add $t0, $zero , $t1
        addi $t2, $zero, 0
        loop_3: ; Loop de linha vermelha no canto esquerdo
                sw $s0, ($t0)
                addi $t0, $t0, 64
                addi $t2, $t2, 1
```

```
                beq $t2, 15, exit_3
                j loop_3
        exit_3:
        ;t0 já está com o valor do final da
        ; linha do canto esquerdo, basta continuar dele

        addi $t2, $zero, 0
        loop_4: ;Loop de linha vermelha no canto inferior
                sw $s0, ($t0)
                addi $t0, $t0, 4
                addi $t2, $t2, 1
                beq $t2, 15, exit_4
                j loop_4

        exit_4:
        addi $t0, $t1, 28 ; t0 recebe a posição inicial em 7*4
        addi $t2, $zero, 0
        loop_5: ; Loop de linha vermelha na parede do meio
                sw $s0, ($t0)
                addi $t0, $t0, 64
                addi $t2, $t2, 1
                beq $t2, 5, exit_5
                j loop_5

        exit_5:
        addi $t0, $t1, 604 ; t0 recebe a posição inicial em 6*4 + 9*64
        addi $t2, $zero, 0

        loop_6: ;Loop de linha vermelha na parede do meio
                sw $s0, ($t0)
                addi $t0, $t0, 4
                addi $t2, $t2, 1
                beq $t2, 8, exit_6
                j loop_6
        exit_6:
        jr $ra
; referencia do pixel:
; (K pontos * 4bytes) é o deslocamento em  x + (K' pontos * 64 )
; é o deslocamento em y  ao somar os pontos temos o valor desejado
```

Abaixo segue a exibição do mapa estático gerado ao final do procedimento.

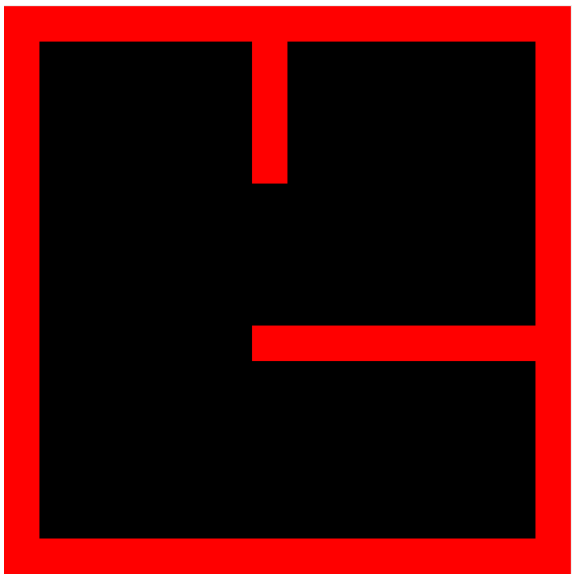

Figura 1. Mapa estático gerado e usado como ambiente, ainda sem móveis.

3.3.4. nMoveis

Este procedimento realiza uma $syscall$ para a impressão da mensagem que solicita a entrada do usuário correspondente ao número de objetos(móveis) que serão gerados e posicionados no mapa. Além da impressão da mensagem, o número entrado pelo usuário é carregado no registrador $s7. Ao final desse procedimento podemos usar o número contido em $s7 para gerar essa quantidade de móveis, o que é feito pelo procedimento $insereMovelF$.

```
1         nMoveis:
2                 li $v0, 4
3                 la $a0, .msgMoveisFixos
4                 syscall
5                 li $v0, 5
6                 syscall
7                 move $s7, $v0 ; recebe o numero de moveis fixos
8                 jr $ra
```

3.3.5. insereMovelF - Gera posições para inserção dos Obstáculos

Como comentado na sessão anterior, este procedimento utiliza a quantidade de objetos(móveis) entrada pelo usuário e que está em $s7. A lógica do procedimento consiste em gerar posições aleatórias para posicionar cada um dos N objetos, porém se essa posição for inválida, ou seja, se já há algum obstáculo nesta posição(parede ou algum outro móvel gerado anteriormente), é necessário gerar uma nova posição para o móvel atual. Esta lógica é implementada pela simulação de um *loop while de alto nível* com o label $loop_mF$, que é chamado caso a posição gerada seja inválida (comparação feita na linha 11 da listagem abaixo). A listagem abaixo mostra o código do procedimento.

```
1  insereMovelF:
2      move $t7, $ra ; salva $ra antes de chamar a proxima função
3      addi $t2, $zero, 0 ; controle de posicoes ja geradas
4      loop_mF:
5              jal rngXY ; define um valor randomico de 0 a 256
6              addi $t4, $zero, 4 ; t4 recebe 4
7              ; Gera o pixel do movel: 4bytes por quadradado * o quadrado desejado
8              mul $t3 , $t4, $s5 ; ex: quadrado 26 em endereço é 26*4
9              add $t0, $t1, $t3
10             ; t0  recebe o pixel inicial em t1 (pixel base) e eh
11             ; somado com t3 para dar o deslocamento do pixel correspondente
12             lw $t3, ($t0) ; le da memoria o valor da posição gerada
13             bne $t3, $s0, gravaF ; se a cor armazenada for diferente, grava no pixel
14             j loop_mF ; se for igual, recalcula pois a posicao eh invalida
15             gravaF:
16                 sw $s0, ($t0) ; preenche o espaço com a posição nova
17                 addi $t2, $t2, 1 ; incrementa o contador
18                 beq $t2, $s7, ex1 ; se contador == numero de móveis, terminamos
19                 j loop_mF ; se não for igual ele retorna para calcular novos pixels
20             ex1:
21                 jr $t7 ; retorna para o calculo do random
```

A linha 2 mantém o contador de móveis já gerados. Este contador é usado na linha 18 dentro de $loop\_mF$ para verificar se ainda precisamos gerar novas posições. As linhas de 5 a 9 geram uma nova posição que pode ser válida ou não. Com isso, verificamos a validade dessa posição simplesmente checando se o conteúdo da posição gerada **já está ocupado por um obstáculo**, ou seja, já contém o vermelho (cor dos obstaculos do

ambiente). Como sabemos que o valor vermelho foi armazenado em $s0 na chamada do procedimento $defCores$, simplesmente o utilizamos para comparar com o valor da posição gerada.

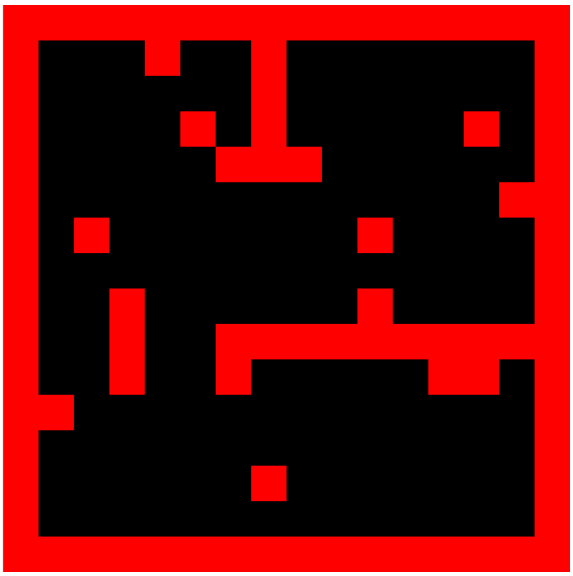

Figura 2. Mapa com os 18 móveis gerados aleatóriamente.

3.3.6. insereRobo - Insere robô no Ambiente

Este procedimento aleatóriamente uma posição inicial para o robô e o insere no mapa. Acompanhamos a posição do robô durante seu percurso no ambiente por meio do registrador $s3. A posição gerada precisa ser tal que não conflite com algum móvel gerado ou alguma parede do ambiente. A imĺementaçãodo procedimento segue abaixo.

```
1  insereRobo:
2          move $t7, $ra
3          loop_r:         ; loop enquanto posicao gerada for invalida
4                  jal rngXY
5                  addi $t4, $zero, 4
6                  mul $t3, $t4, $s5
7                  add $t0, $t1, $t3
8                  lw $t3, ($t0)
9                  bne $t3, $s0, c1
10                 j loop_r ;
11                 c1:
12                     bne $t3, $s1, gravr
13                     j loop_r
14                 gravr: ; grava posicao do robo e retorna 'na pilha'
15                     sw $s3,($t0)
16                     jr $t7
```

As linhas 4 a 8 são responsáveis por gerar uma posição aleatoriamente, como essa posição pode coincidir com algum outro objeto no mapa, verificamos se o a cor no mapa, da posição gerada, é igual a vermelho (objetos no mapa são vermelhos). Caso a igualdade, recalculamos uma nova posição (jump das linhas 10 e 12). Se a posiçaõ gerada é uma posição válida gravamos essa posição em $s3 (linha 15).

3.4. Movimentação do Robô

Abaixo segue o procedimento que implementa a movimentação do robô pelo ambiente. Como comentado anteriormente o robô possui uma **heurística de busca gulosa**, onde o robô sempre escolhe visitar espaços onde ele menos esteve, na esperança de que estes sejam próximos de espaços onde ele nunca esteve. Para isso ele necessita de uma foma de marcar os lugares onde esteve e quantificar a taxa de vezes que visitou aquele determinado ponto. Esse comportamento é implementado da seguinte maneira:

1. Todos os pontos vazios no mapa começam com a cor **preta**. Iniciamos um contador de posições restantes como o total de posições pretas.
2. Ao visitar um ponto no mapa o robô verifica a cor do ponto, caso for preta o contador é decrementado, sinalizando que mais um ponto foi visitado pela primeira vez. Como esse ponto foi visitado o robô então incrementa a cor desse ponto para indicar que esse é um ponto recentemente visitado. Portanto **pontos que foram visitados menos vezes possuem uma cor de valor menor e pontos recentemente visitados possuem uma cor de valor maior**.
3. Para se movimentar o robô escolhe o ponto vizinho que possui a menor cor, ou seja ele faz uma **escolha gulosa** pelo vizinho mais promissor que é o que foi menos visitado, na esperança de que os vizinhos dele também foram menos visitados(alguns possivelmente nunca visitados).
4. O robô pára quando seu contador atinge o valor 0, significando que todas as posições pretas(validas) já foram visitadas.

É possível demonstrar que esta heurística de busca reduz o espaço de busca(e consequentemente o tempo de execução) por um fator exponencial em relação à busca totalmente aleatória, isso à medida que o tamanho do problema cresce(dimensão do mapa). Abaixo segue o procedimento que implementa o comportamento do robô. Note que o procedimento é chamado para cada passo do robô, ou seja, para cada movimentação de um ponto para algum ponto vizinho.

```
moveRobo: ; seleciona um candidato a pixel na vizinhança
        ; t0 vai estar com a posição do escolhido
        subi  $s4, $t0, 68 ; s4 recebe ponto da diagonal esquerda superior
        move $s5, $t0 ; carrega valor do ponto a cima em s5
        subi $s5, $t0, 64 ; s5 recebe o ponto superior
        lw  $t3, ($s4) ; recebe o conteudo de s4
        lw  $t4, ($s5) ; recebe o conteudo de s5
        slt $t7, $t3, $t4 ; se s4<s5 t7 recebe 1
        beq $t7,1, prox1
        ;se nao for menor
        move $s4, $s5 ; s4 recebe s5
        prox1:
            ;s4 vai estar com o menor valor
            move $s5, $t0
            subi $s5, $t0, 60 ; recebe o ponto da diagonal superior direita
            lw  $t3, ($s4) ; recebe o conteudo de s4
            lw  $t4, ($s5) ; recebe o conteudo de s5
            slt $t7, $t3, $t4 ; s4 eh menor que s5
            beq $t7,1, prox2
            ; se nao for menor
            move $s4, $s5 ; t7 recebe s5
        prox2:
            move $s5, $t0
            subi $s5, $t0, 4 ; recebe o ponto da esquerda
            lw  $t3, ($s4) ; recebe o conteudo de s4
            lw  $t4, ($s5) ; recebe o conteudo de s5
            slt $t7,$t3, $t4
```

```
28              beq $t7,1,prox3
29              move $s4, $s5 ; t7 recebe s5
30          prox3:
31              move $s5, $t0
32              addi $s5, $t0, 4 ; s5 recebe o ponto da direita
33              lw  $t3, ($s4) ; recebe o conteudo de s4
34              lw  $t4, ($s5) ; recebe o conteudo de s5
35              slt $t7,$t3, $t4
36              beq $t7,1, prox4
37              move $s4, $s5 ; t7 recebe s5
38          prox4:
39              move $s5, $t0
40              addi $s5, $t0, 60 ; s5 = diagonal esquerda inferior
41              lw  $t3, ($s4) ; recebe o conteudo de s4
42              lw  $t4, ($s5) ; recebe o conteudo de s5
43              slt $t7,$t3, $t4
44              beq $t7,1,prox5
45              move $s4, $s5 ; t7 recebe s5
46          prox5:
47              move $s5, $t0
48              addi $s5, $t0, 64 ; s5 recebe o ponto debaixo
49              lw  $t3, ($s4) ; recebe o conteudo de s4
50              lw  $t4, ($s5) ; recebe o conteudo de s5
51              slt $t7,$t3, $t4
52              beq $t7,1,prox6
53              move $s4, $s5 ; t7 recebe s5
54          prox6:
55              move $s5, $t0
56              addi $s5, $t0, 68 ; s5 recebe o ponto da diagonal direita inferior
57              lw  $t3, ($s4) ; recebe o conteudo de s4
58              lw  $t4, ($s5) ; recebe o conteudo de s5
59              slt $t7,$t3, $t4
60              beq $t7,1,prox7
61              move $s4, $s5 ; t7 recebe s5
62          j prox7
63          prox7:
64              lw $s2, ($s4) ; carrego o conteudo da variavel selecionada em s2
65              beq $s2, 0x00000000, dim
66              j continue ; se nao for igual a preto
67          dim:
68              subi $t8,$t8,1 ; se for diminui um do contador
69          continue:
70              sw $s3, ($s4); pinto a posicao selecionada de azul
71              addi $s2, $s2, 50 ;incrementa a cor da posicao atual
72              sw $s2, ($t0); pinta a posicao atual com o incremento
73              move $t0,$s4 ; t0 recebe a nova posicaorobo
74      jr $ra
```

A lógica das linhas de 3 a 11 são repetidas 8 vezes(uma para cada vizinho candidato à seleção). Primeiro calculamos o endereço do primeiro, feito isso carregamos a cor de sua posição e armazenamos em um registrador temporário $t3. Tendo agora a cor de um vizinho, o algoritmo segue verificando se essa cor é menor do que a cor dos outros vizinhos(pontos candidatos a seleção para realizar o movimento), repetindo esse conjunto de instruções para todos os vizinhos e caso algum deles possuir cor de valor menor que o $t3, atualizamos esse registrador com a cor do novo menor vizinho, além de manter sua posição em $s4.

Após ter escolhido o vizinho de menor cor e ter o endereço de memória dele como o próximo local que o robô visitará, as linhas de 66 a 74 realizam de fato a movimentação do robô. A linha 66 verifica se o vizinho selecionado é da cor preta, caso for há um *jump* para o label *dim* que decrementa o contador de posições há se visitar. As linhas seguintes (71 a 73) incrementam o valor da cor da posição escolhida em 50 unidades(valor hexa-

decimal) e a ultima porém não menos importante linha 74 move o robô atualizando o registrador que armazena sua posição $t0 com o endereço de memória do vizinho selecionado $s4.

4. Execução

Esta sessão mostra algumas execuções do Software do robô com diferentes quantidades de móveis e diferentes posições iniciais.

4.1. Executando no emulador MARS

Primeiramente conectamos o *Bitmap Display* do MARS com as configurações:

- Unit width in Pixels: 32
- Unit weight in Pixels: 32
- Display width in Pixels: 512
- Display weight in Pixels: 512

Após isso Conectamos o *Bitmap Display* ao MIPS e executamos o código.

O programa irá solicitar a quantidade de objetos a inserir no mapa, após a entrada do usuário o robô será aleatoriamente posicionado no mapa e fará seu percurso passando por todos os pontos ao menos uma vez.

4.2. Exemplo de Execuções

5. Conclusão

O software do robô aspirador foi verificado no Mars e obtém a trajetória final correta do robô para diversas situações iniciais. A heurística de busca proposta otimiza o tour do robô de forma que ele visita um menor número vezes posições que ele já visitou e tende à seguir caminhos pouco visitados que levam a um ponto ainda não visitado.

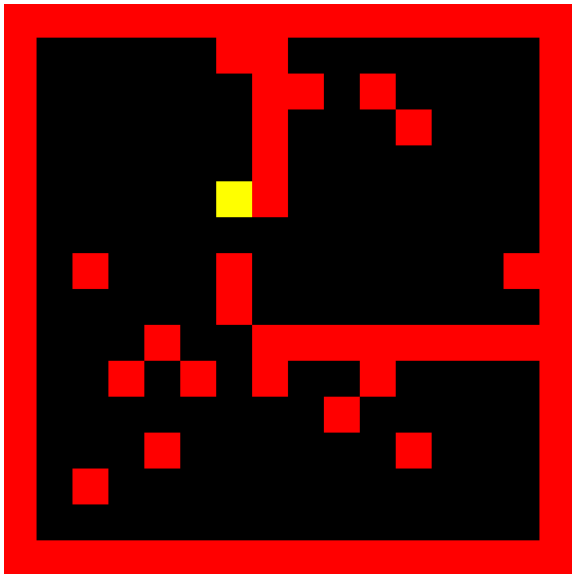

Figura 3. Execução 01 - Mapa inicial com 18 móveis.

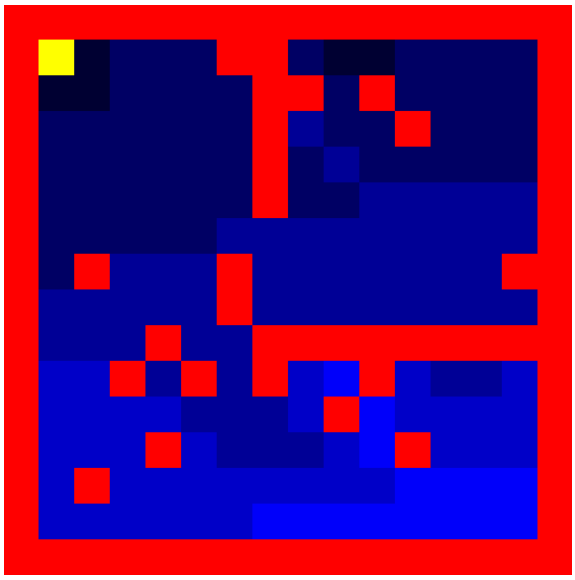

Figura 4. Execução 01 - Mapa após tour do robô.

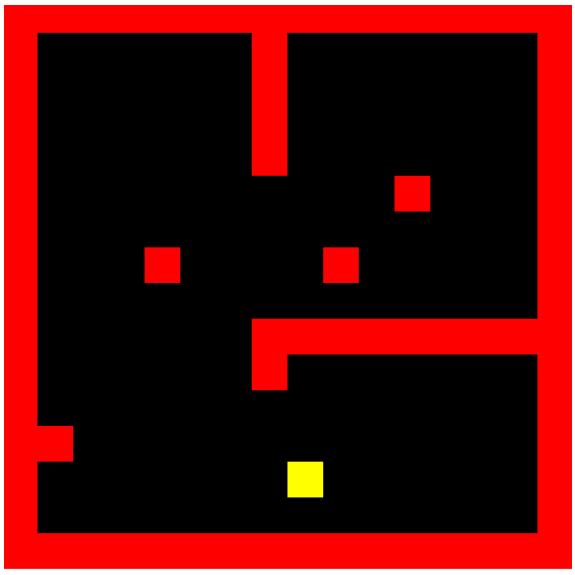

Figura 5. Execução 02 - Mapa inicial com 4 móveis.

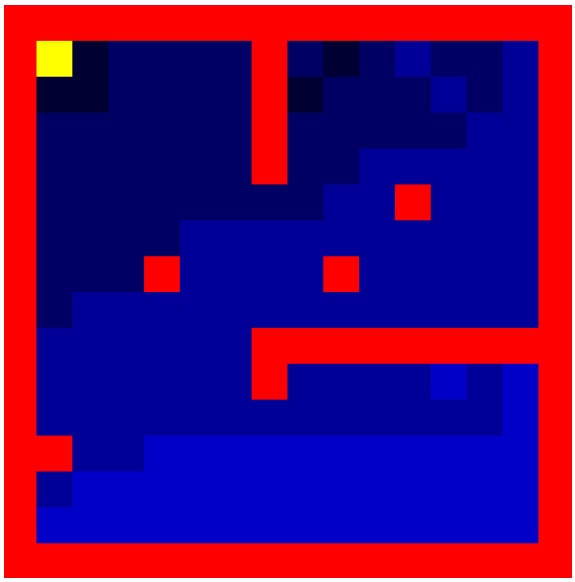

Figura 6. Execução 02 - Mapa após tour do robô.