OTIMIZAÇÃO DE ALGORITMO DE RESOLUÇÃO DE LABIRINTO PARA MICROMOUSE DE BAIXO CUSTO

Francisco Marcolino Rodrigues Filho*, Otacílio da Mota Almeida*

* *Universidade Federal do Piauí*
Departamento de Engenharia Elétrica
Teresina, Piauí, Brasil

Emails: fmarcolino@live.com, otacilio@ufpi.edu.br

Abstract— Micromouse is an old competition. From the 70's to the present day there are several competitions of the sport all over the world. Every year supporters and fans of this type of robot try to create more efficient algorithms in order to win the championship, which consists of giving prizes to the robots that solve the labyrinth as quickly as possible. There are several resolution algorithms, however, the most commonly used is flood fill. It is based on making the surface of the platform embossed, giving numbers for each cell. From this, the robot tends to descend this surface to the lowest point, in the form of swirls, as does the water in the river in its natural course. Sometimes the mini-robot differential with a microprocessed system is able to run extremely fast codes, which is important so that it has agility in the process of searching for the smallest path. However, beginners in this mode end up acquiring micromouses with limited RAM and it is important to optimize the algorithm so that you use it with caution. With the use of the Falcon C ++ program, intelligent algorithm of labyrinth resolution, in C language, optimized for microcontrollers of low cost was developed. It was possible to use a single data structure to store both the shortest path and the back path, reducing the use of RAM for storage of distances by 50Therefore, this work aims to develop the intelligent algorithm of labyrinth resolution, in C language, optimized for microcontrollers at low cost, important step for the project micromouse in question.

Keywords— Micromouse, Intelligent algorithm, maze, Flood fill, Robotics.

Resumo— Micromouse é uma competição antiga. Desde os anos 70 até os dias atuais existem várias competições da modalidade em todo o mundo. Todos os anos simpatizantes e fãs deste tipo de robô tentam criar algoritmos mais eficientes a fim de ganhar o campeonato, o qual consiste em dar prêmios aos robôs que resolvem o labirinto o mais rápido possível. Existem diversos algoritmos de resolução, porém, o mais utilizado é o *flood fill*. Ele é baseado em tornar a superfície do tablado em *relevo*, dando números para cada célula. A partir disso, o robô tende a descer esta *superfície* até o ponto mais baixo, em forma de redemoinhos, como faz a água no rio em seu curso natural. Às vezes o mini-robô diferencial com um sistema microprocessado é capaz de rodar códigos extremamente rápidos, quesito importante para que tenha agilidade no processo de busca de menor caminho. Porém, iniciantes nesta modalidade acabam adquirindo micromouses com memória RAM limitada e é importante a otimização do algoritmo para que a utilize com cautela. Com o uso do programa Falcon C++, foi desenvolvido algoritmo inteligente de resolução de labirinto, em linguagem C, otimizado para microcontroladores de baixo custo. Foi possível a utilização de uma única estrutura de dados para armazenar tanto o menor caminho de ida como também o da volta, reduzindo 50% do uso de memória RAM para armazenamento das distâncias. Portanto, este trabalho tem como objetivo o desenvolvimento do algoritmo inteligente de resolução de labirinto, em linguagem C, otimizado para microcontroladores de baixo custo, passo importante para o projeto micromouse em questão.

Palavras-chave— Micromouse, Algoritmo inteligente, labirinto, Flood fill, Robótica.

1 Introdução

Micromouse é uma competição antiga. Desde os anos 70 até os dias atuais existem várias competições da modalidade em todo o mundo (Li et al., 2010).

1.1 Um breve historico do Micromouse

O micromouse é um concurso onde envolve robôs (com mesmo nome) com dimensões limitadas para resolver um labirinto de 16x16 células. Este conceito foi introduzido pela revista do *IEEE Spectrum* em 1977, sendo que atualmente ela existe em vários países que fazem a competição. Este robô tem um tempo limitado para resolver um labirinto de 256 células, sendo que a configuração do labirinto é revelada somente na competição (Li et al., 2010).

Os primeiros robôs eram grandes e *desengonçados*, comparados com os mais atuais. Os micromouses nada se pareciam com *ratos*, mas com aves. Os sensores de paredes ficavam sob as *asas*, o que deixava-os lento, uma vez que com centro de gravidade elevado e largo as chances de tombamento eram altas.

À medida que a tecnologia avançava, os robôs também se mordenizaram. Dispositivos compactados *SMD*, tais como microcontroladores, etc, em conjunto com tecnologia de construção e solda de tais dispositivos, o robô tornou-se mais compacto e rápido. Tanto que já é possível o robô mover-se em diagonal ao invés de zigue-zague em células quadradas de 180 mm de largura.

1.2 Micromouse como ferramenta de estímulo

Segundo Silva et al. (2015), o projeto micromouse é considerado mais uma ferramenta para estimular o interesse em alunos na área das Ciências,

Tecnologia, Engenharia e Matemática, promovendo competências consideradas pré-requisitos para carreiras nas TI. Lopez et al. (2015) acreditam também que o mini robô móvel autônomo é ideal para dar oportunidades de o estudante expandir seus conhecimentos em diversas áreas.

Para os alunos do curso de Engenharia Elétrica da UTAD (Portugal), participar da competição é uma experiência de grande aprendizado. O contato com diferentes softwares de programação e com a robótica foi uma forma de aprofundar os conhecimentos obtidos no curso e despertar o interesse dos alunos para áreas como controle e automação (Silva et al., 2015).

1.3 As regras do concurso Micromouse

A função básica do Micromouse é sair de um dos cantos do labirinto de 16x16 células, de cuja área é de 18 cm^2 cada, e chegar ao centro do mesmo. Isto se chama corrida (*run* - em inglês). É contado este tempo. O tempo da corrida do centro à célula de partida é desconsiderado. O micromouse é pontuado de acordo com três parâmetros: velocidade, eficiência em resolver labirinto, e confiabilidade do Micromouse (Li et al., 2010).

Além disso, existe um tempo de conhecimento do labirinto (primeira corrida). O Micromouse tem até 15 minutos para fechar a *corrida*. Depois deste, ele poderá tentar várias vezes a corrida a tentar diminuir o seu tempo (Li et al., 2010).

O micromouse deve ser totalmente autônomo. Caso necessite de alguma assistência no meio da corrida, o mesmo é penalizado por *toque*. O micromouse tem que ter as seguintes características para uma boa competição (Dai et al., 2015):

- estável e rápido
- precisão
- memória de caminho mais curto
- habilidade de resolver qualquer labirinto

Após 2010, para aumentar o nível do desafio, as regras para tamanho mudaram. No Japão, a competição avançou de tal maneira que os micromouses já podem percorrer labirintos de 32x32 células, competições especiais chamadas de *half-size micromouse* (Su et al., 2013). Na Figura 1, é mostrada a competição naquele país.

1.4 Algoritmos de resolução

Todos os anos simpatizantes e fãs deste tipo de robô tentam criar algoritmos mais eficientes a fim de ganhar o campeonato, o qual consiste em dar prêmios aos robôs que resolvem o labirinto o mais rápido possível.

Ao buscar referências bibliográficas e arquivos que contenham tais algoritmos inteligentes para resolução de labirinto, foi encontrado um simulador. Neste, no *Micro Mouse Maze Editor and Simulator*, uma ferramenta em *Java* que mostra como o robô se comporta em diversos labirintos - inclusive alguns já foram labirinto oficial de competição, são encontrados quatro algoritmos. São eles:

Figura 1: Torneio internacional Micromouse *half-size* no Japão

Fonte: (Su et al., 2013)

- `Left Wall Follower`: Seguidor de paredes à esquerda.
- `Right Wall Follower`: Seguidor de paredes à direita.
- `Treumax`: Algoritmo força bruta - visita todas as células do labirinto.
- `Flood Fill`: O mais utilizado - simples e eficiente para este tipo de competição.

Os seguidores de parede são os algoritmos mais simples, pelo fato de se existir vários caminhos, ele segue somente o caminho que foi programado, seja à esquerda ou à direita. Segundo Yadav et al. (2012), algoritmos deste naipe não conseguem lograr êxito em labirintos complexos.

O algoritmo Tremaux é um algoritmo que é eficaz, pois resolve muito bem o labirinto, mas o inconveniente dele é o fato de ele percorrer todas as células do labirinto.

Dos algoritmos apresentados o Flood Fill foi o que atendeu as melhores expectativas pelo fato de não precisar percorrer todo o labirinto.

2 O *Flood Fill*

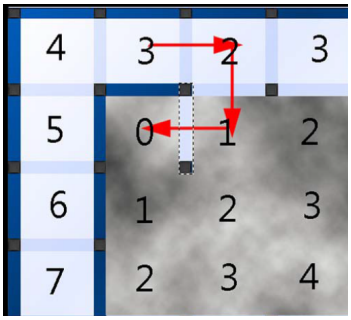
O Flood Fill é um algoritmo que cria uma matriz pré-programada com as dimensões do labirinto, sendo que os números são colocados de uma forma que o labirinto se pareça com um redemoinho, o início tem a numeração mais alta, e o final (chegada) é numerado com zero. A ideia desse algoritmo é percorrer as células do maior para o menor valor (Yadav et al., 2012).

Segundo Silva et al. (2015), o mais utilizado e um dos mais eficientes é o *flood fill* clássico. Ele é baseado em tornar a superfície do tablado em *relevo*, dando números para cada célula. A partir disso, o robô tende a descer esta *superfície* até o ponto mais baixo, em forma de redemoinhos, como faz a água no rio em seu curso natural. O destino sempre tem a numeração 0; uma célula com valor 1 está a um passo do alvo. Se tiver valor 4, está a quatro passos para o destino (Silva et al., 2015).

Um robô seguidor de linha também foi programado com algoritmo flood fill. ROSA et al. (2015) conseguiram programar o mesmo algoritmo, mas para labirintos sem medida prévia. Ela utilizou-se de mini robô seguidor de linha como guia orientadora no labirinto, sistema inteligente diferente do que se planeja neste trabalho.

Já Cai et al. (2012) modificaram o algoritmo *flood fill* para ter melhor performance do que o tradicional. Segundo ele, o algoritmo *flood fill ET*, assim designado, já possui uma inteligência a mais. Ele tenta prever se nas próximas células desconhecidas se há possibilidade de ter uma possível parede, e assim, ele não terá que ir até a célula para verificar isso (Figura 2). Esta inteligência reduziu os tempos de curvas e números de visitas de células do labirinto.

Figura 2: *Flood fill Expected Toll* - Com expectiva de penetração



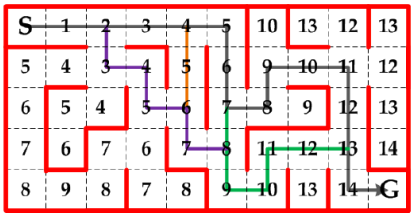
Fonte: adaptado de Cai et al. (2012)

Porém, nas competições mais atuais, ao realizar corrida, os robôs percorrem em *diagonal* (Figura 3) ao invés de realizar contornos em forma de zigue-zague, como mostra a Figura 4. Com isto, os robôs evitam perder velocidade nas curvas, o que melhorou e muito o tempo de corrida (Su et al., 2013). Este algoritmo é o *diagonal flood fill*, considerado melhor, porém não será abordado neste trabalho.

2.1 Princípio do Flood Fill

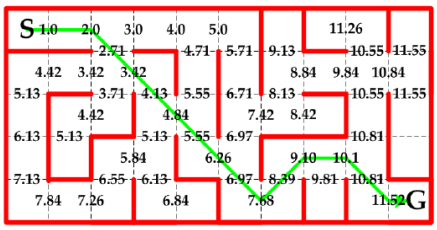
O algoritmo flood fill interpreta o labirinto como um plano cartesiano, como mostra a Figura 5. Cada coordenada é responsável por acessar uma estrutura de dados. A coordenada X representa a linha e a Y representa a coluna da célula. As

Figura 3: Flood fill clássico



Fonte: adaptado de Su et al. (2013)

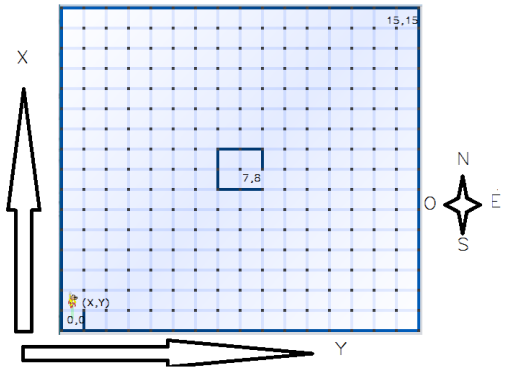
Figura 4: Flood fill diagonal



Fonte: adaptado de Su et al. (2013)

setas mostram o sentido de crescimento dos números das coordenadas. Também é padronizada a referência de orientação. Ou seja, se o robô está se movimentando da célula 0,0 para a célula 1,0, então ele está andando para o Norte.

Figura 5: Interpretação do labirinto para o flood fill



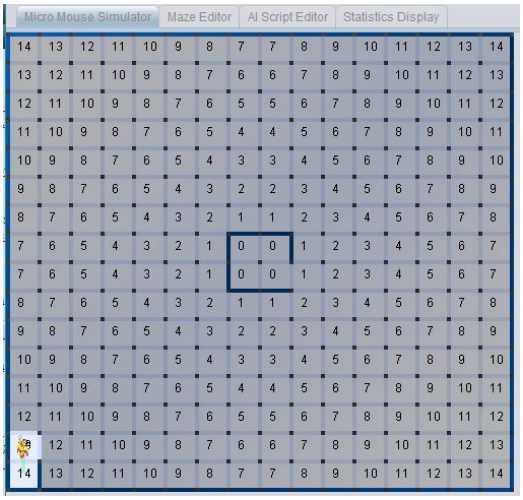
Fonte: do autor

Cada célula possui informações das paredes. Por exemplo, na coordenada 15,15 (Figura 5), há parede ao norte e ao leste. Então, estas informações podem ser acessadas diretamente da célula, ou seja, as variáveis célula(15,15).wall[N] e célula(15,15).wall[E] retornarão verdadeiras, enquanto as variáveis célula(15,15).wall[O] e célula(15,15).wall[S] retornarão falsas. Então, à medida que o robô vá percorrendo o labirinto, ele vai atualizando as informações das paredes das células, além de atualizar também a variável Checado.

Então, o algoritmo encara o labirinto como se ele não tivesse paredes e enumera as células ini-

cialmente conforme mostrado na Figura 6. Estes valores são o número mínimo de passos até então para chegar ao alvo (valor zero). Um passo moverá o robô em sua célula atual para a célula com a distância menor que a distancia da célula atual. Por exemplo, adicionando uma unidade da coordenada X, e mantendo a mesma coordenada para Y, e verificando que a distância da célula((x+1),y) é menor que o valor da distância da célula(x,y), o robô terá de mover-se verticalmente para o Norte até a próxima célula. A princípio, o robô sempre procurará adentrar para a célula que tem o menor valor de distância até encontrar a célula de destino (contém distância zero).

Figura 6: Numeração das distâncias sobre o labirinto



Fonte: do autor

Como foi mencionado anteriormente, o número da distância em cada célula é levado em conta que o robô encara as áreas desconhecidas como aberto, ou seja, sem paredes. Então, conforme mostra a Figura 2, o valor da célula(3,1) é valor 3 porque não há conhecimento da parede Leste da célula(2,1). Até aquele momento, naquela coordenada, o robô seguiria o percurso das setas, uma vez que seria o número de passos para chegar ao alvo. Esta concepção é levada para todo o labirinto.

Segundo as regras da competiçao, as quatro células centrais tem a distância dada como zero. Então a partir destas células, é aplicada a regra das distâncias das células vizinhas para determinar os valores de cada célula.

3 A receita

O algoritmo é feito conforme os passos abaixo.

- Inicialmente o labirinto é encarado como se não existissem paredes.
- Enumerar as células com a distância até o alvo, que tem a distância igual a 0.
- Repetir ate que se chegue à célula-alvo.
- Ir para a célula vizinha com menor distância
- Rodar o algoritmo de atualização das distâncias

O algoritmo de atualização das distâncias é não recursivo. Ele utiliza de pilhas para armazenamento do endereço das células empilhadas. Desta forma, não é necessária a recursividade, que aumentaria o consumo de memória.

Alguns passos são necessários para este algoritmo de atualização.

- Verificar a pilha se ela está vazia
- Se alguma parede foi descoberta:
  - Guardar na pilha a célula atual e as células próximas às paredes.
- Enquanto a pilha não esvazia
  - retirar do topo da pilha o endereço da célula guardado
  - verificar se a distância mínima do vizinho aberto àquela célula + 1 é a distância da célula atual
  - Se isto não é verdade, deve-se modificar o valor da célula atual para o valor mínimo da distância das células vizinhas + 1 e guardar todos os vizinhos abertos para a pilha.

4 O problema de memória RAM

Microcontroladores de baixo custo possuem pouca memória, o que os tornam não recomendáveis a execução do algoritmo, além disso ainda compartilhar memória para rodar o controle do robô. Pilhas são a estrutura básica de dados do *Flood Fill*. A todo momento o algoritmo guarda e retira a informação da pilha de forma não-recursiva que pode chegar a centenas de empilhamentos, a depender da complexidade do labirinto.

Cada célula contém estruturas de dados para armazenar as seguintes informações:

- `distância`: Esta variável, do tipo inteiro de 16 bits, contém o número de passos para o robô chegar ao alvo.
- `paredes`: Esta variável, na verdade é um vetor do tipo booleano de quatro posições. Cada índice, de 0 a 3, armazena, respectivamente, a informação das paredes descobertas pelo robô ao norte, leste, sul e oeste. Exemplo: paredes[0] é 1 quando tem parede ao norte ou 0 quando não tem parede ao norte.
- `Checado`: Esta variável do tipo booleana é alterada para verdadeiro, caso o robô tenha visitado esta célula. Importante para evitar consumo de energia dos sensores de distância.

Para o algoritmo em questão, como o robô precisa sair do canto ao centro e voltar, são necessárias mais memória para armazenar o caminho de ida e de volta. Além disso, outra estrutura de dados é necessária: a do robô. Ela é responsável por armazenar a sua posição cartesiana diante do labirinto, orientação e sentido. Porém, não há necessidade de tanta preocupação com esta.

Geralmente os microcontroladores tem bom desempenho, desde dezenas até centenas de MIPS. A proposta é promover a construção do algoritmo em prol da menor utilização de memória RAM, porém com custo computacional maior, porém, com pouco impacto e resultados equivalentes ao algoritmo tradicional já mostrado nas seções anteriores.

5 Metodologia

Um dos melhores algoritmos inteligentes para resolução de labirintos, segundo Yadav et al. (2012), decidiu-se optar pelo desenvolvimento do algoritmo não-recursivo *Flood Fill* em linguagem C. Foi escolhida esta linguagem porque ela é utilizada comumente em microcontroladores de baixo custo. Além disso, a linguagem C é a mais enxuta próxima às outras linguagens de alto nível e faz com que utilize menos memória flash, quesito importante para otimizar o uso do mesmo.

Portanto, este trabalho tem como metodologia realizar e simular, em um primeiro momento, o algoritmo inteligente de resolução de labirintos desconhecidos em linguagem C num programa IDE Falcon C++, para utilizá-lo em micromouse de baixo custo. A proposta é de utilizar somente uma mesma estrutura de dados para armazenar tanto o caminho da ida do robô como também o da volta.

6 Resultados e Discussões

O algoritmo foi depurado através de símbolos impressos em janela do *prompt do windows*. Através dele, foi possível a analise do percurso do robô. Abaixo são mostrados alguns símbolos básicos essenciais para depurações.

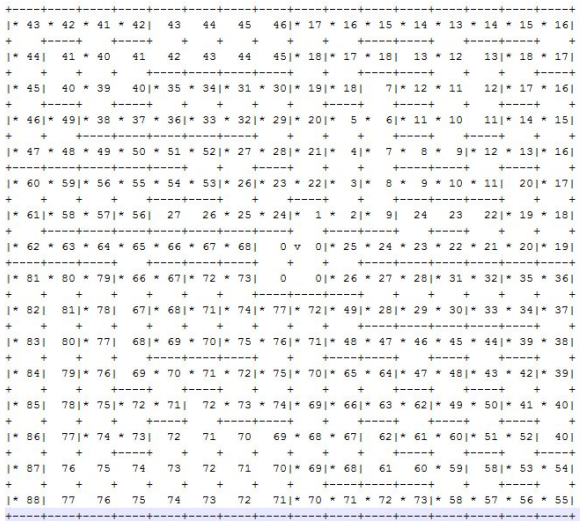
```
+----+----+----+
|*  5 *  4 V  3|
+----+----+    +
|   0    1    2|
+----+----+----+
```

O símbolo `'+'` foi utilizado para as quinas. O `'---'`, para as paredes Norte/Sul. Já o símbolo `'|'`, para indicar as paredes Leste/Oeste. E o asterisco, para indicar que a célula já foi visitada. Foram utilizados também os símbolos `<, >, ^ , V` para mostrar a orientação do micromouse.

No algoritmo proposto, a cada corrida (tanto da IDA como da VOLTA) as distâncias das células são reinicializadas como se o labirinto não tivesse nenhuma parede. Com esta estratégia, utilizando a mesma estrutura de memória, economiza 50% de memória ram neste quesito.

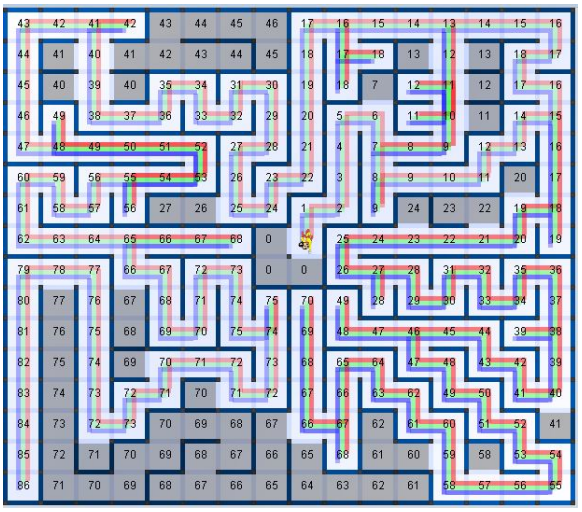
A Figura 7 já mostra a ida do robô. Em comparação com o percurso do algoritmo do simulador *Micro Mouse Maze Editor and Simulator* da Figura 8, são bastante idênticos ambos os percursos. As causas desta pequena diferença tem a ver com a convenção de dar preferência em seguir linha reta ao invés de realizar a curva, em vizinhanças de mesmo número.

Figura 7: Percurso do robô no labirinto *Seoul* com algoritmo criado na primeira corrida *real*



Fonte: do autor

Figura 8: Percurso do robô no labirinto *Seoul* com algoritmo do *Micro Mouse Maze Editor and Simulator*



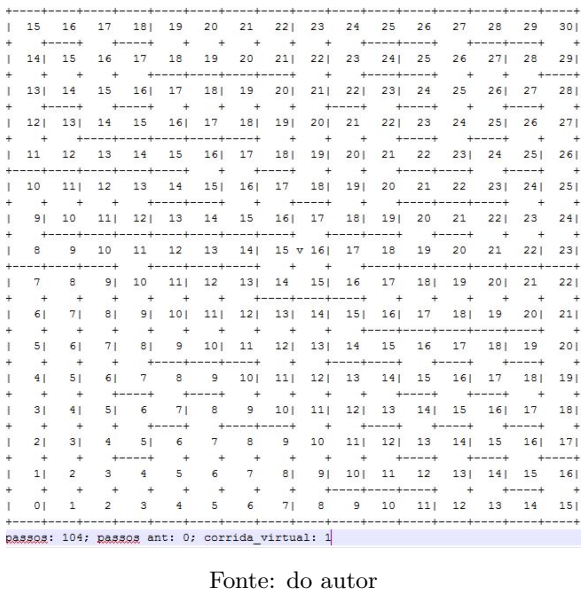
Fonte: do autor

Então, são redefinidas as distâncias das células e inicia-se a *varredura virtual* do robô. A condição de parada é quando o número de passos

da corrida anterior se torna idêntica ao número de passos da última tentativa. Assim, o robô poderá sair da célula central à celula de partida seguramente da menor distância possível para voltar.

Para o labirinto em questão, foram necessárias 5 corridas virtuais para este *primeiro run*. A tendência é que, conforme vá aumentando o número de corridas *reais*, o número de corridas virtuais caia. A Figura 9 mostra a redefinição das distâncias para permitir a volta do robô. A figura 10 mostra a terceira corrida virtual. Já a figura 11 mostra a última corrida virtual, com detalhe no número de passos. Conforme vá utilizando o algoritmo *flood fill* e o algoritmo de atualização das distâncias para corridas *virtuais*, o robô terá o seu menor caminho para chegar à célula de partida de fato.

Figura 9: Atualização das distâncias das células para o robô voltar à celula de partida - corrida virtual 1



Fonte: do autor

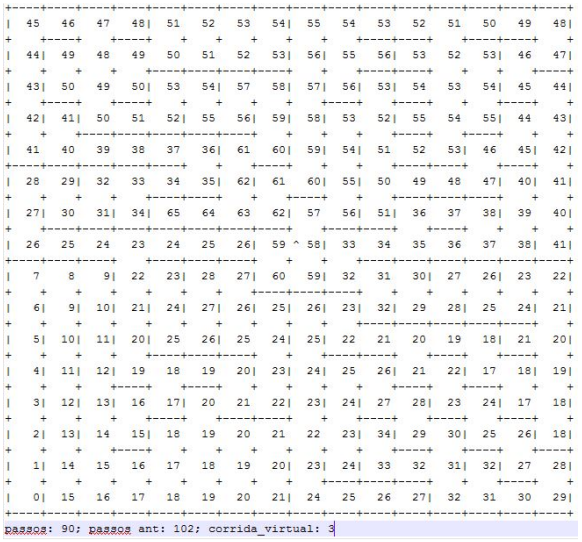
O processo se repete na preparação da IDA do robô à célula de destino. Desta forma, conseguiu-se usar somente uma estrutura de dados que armazena tanto o caminho de ida como também o caminho de volta.

A eficiência do algoritmo permanece intacta. Após 3 corridas *reais*, tanto o algoritmo proposto como o algoritmo do simulador *Micro Mouse Maze Editor and Simulator* apresenta a mesma distribuição dos números das células, como mostram as Figuras 12 e 13.

7 Conclusões

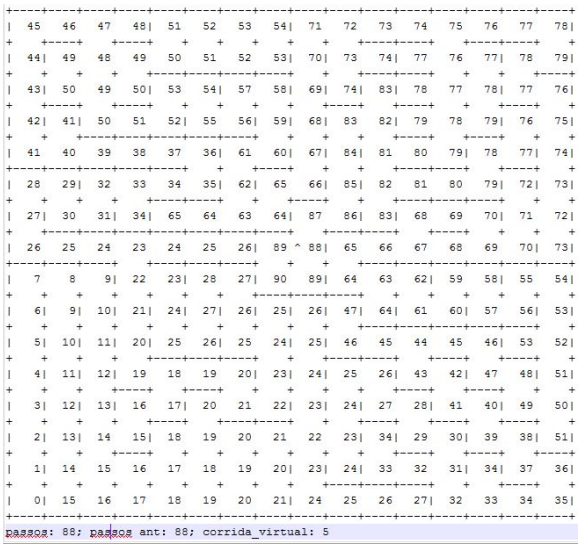
A competição micromouse é interessante, porém, iniciantes começam com micromouses de baixo custo (com microcontroladores da ATMEGA, PIC e ST), o que dificulta o uso do algoritmo *flood fill*, justamente por ele requerer muita memória RAM.

Figura 10: Atualização das distâncias das células para o robô voltar à celula de partida - corrida virtual 3



Fonte: do autor

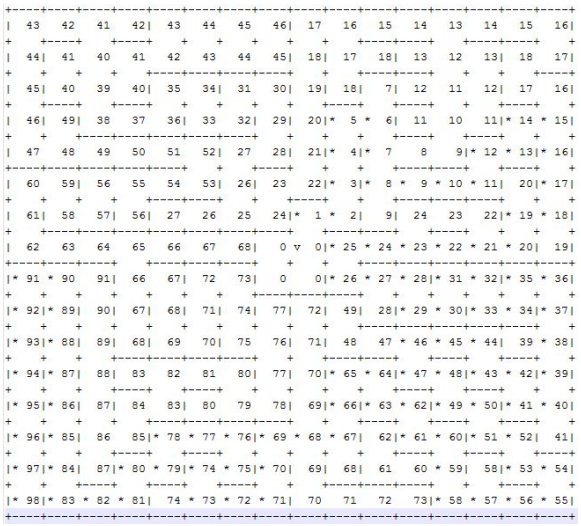
Figura 11: Atualização das distâncias das células para o robô voltar à celula de partida - corrida virtual 5



Fonte: do autor

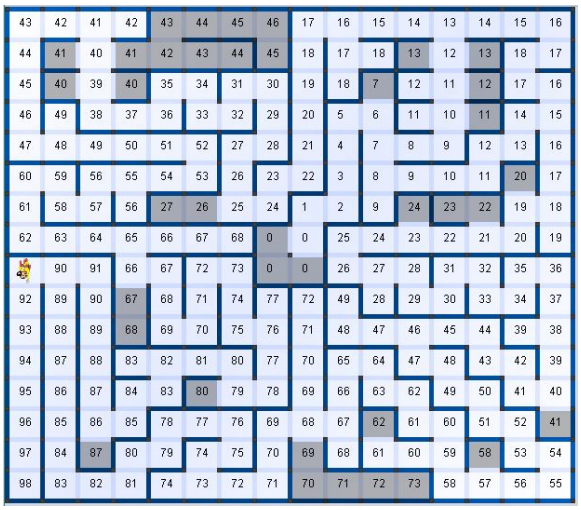
O algoritmo construído foi otimizado, utilizando somente uma estrutura de dados para armazenar as distâncias tanto de ida como de volta. E não houve mudanças na eficiência do algoritmo, visto que ambos os simuladores mostram os mesmos percursos. Já para o caminho de volta, como já foi visto, as distâncias são redefinidas, uma vez que o novo alvo é a célula inicial (coordenadas 0,0). Porém, as informações das paredes descobertas ainda permanecem na memória, bastando apenas realizar varreduras *virtuais* para atualizar as distâncias. Isto é feito enquanto o robô permanece parado na célula de destino. Assim,

Figura 12: Labirinto *Seoul* após 3 corridas *reais*



Fonte: do autor

Figura 13: Labirinto *Seoul* após 3 corridas *reais* no *Micro Mouse Maze Editor and Simulator*



Fonte: do autor

quando as atualizações se completam, o robô poderá deslocar-se para as células de menor distância. O processo se repete quando o robô chega à célula de distância zero. Haverá redução de uso de memória RAM e um custo computacional maior, porém isto acontece enquanto o robô permanece parado e fora do tempo de corrida, não prejudicando o desempenho do robô em uma eventual competição. Embora com maior custo computacional para realização das *corridas virtuais*, há viabilidade do uso deste algoritmo para microcontroladores de pouca memória RAM, que foi otimizada para uso duplo.

Agradecimentos

Agradeço ao PET POTÊNCIA do curso de Engenharia Elétrica da UFPI.

Referências

Cai, Z., Ye, L. and Yang, A. (2012). Floodfill maze solving with expected toll of penetrating unknown walls for micromouse, *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, pp. 1428–1433.

Dai, S., Zhao, B., Li, Z. and Dai, M. (2015). Design and practice from the micromouse competition to the undergraduate curriculum, *2015 IEEE Frontiers in Education Conference (FIE)*, pp. 1–5.

Gupta, B. and Sehgal, S. (2014). Survey on techniques used in autonomous maze solving robot, *2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence)*, pp. 323–328.

Li, X., Jia, X., Xu, X., Xiao, J. and Li, H. (2010). An improved algorithm of the exploring process in micromouse competition, *2010 IEEE International Conference on Intelligent Computing and Intelligent Systems*, Vol. 2, pp. 324–328.

Lopez, G., Ramos, D., Rivera, K., del Valle, K., Rodriguez, A. and Rivera, E. I. O. (2015). Micromouse: An autonomous robot vehicle interdisciplinary attraction to education and research, *2015 IEEE Frontiers in Education Conference (FIE)*.

ROSA, A. S., LIMA, H. R. K. and SALOMÃO, J. M. (2015). Um algoritmo para a solução de labirintos com uso de um robô móvel, *SBAi 2015* .

Silva, S., Soares, S., Valente, A., Barradas, R. and Bartolomeu, P. (2015). A iniciativa micromouse e o estímulo ao saber tecnológico no ensino pré-universitário.

Su, J. H., Huang, H. H. and Lee, C. S. (2013). Behaveior model simulations of micromouse and its application in intelligent mobile robot education, *2013 CACS International Automatic Control Conference (CACS)*, pp. 511–515.

Yadav, S., Verma, K. K. and Mahanta, S. (2012). The maze problem solved by micro mouse, *2012 IEEE Frontiers in Education Conference (FIE)*.