Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Departamento de Engenharia Elétrica

Robôs Móveis Autônomos

Projeto de Implementação

Relatório

Prof. Dr. Kelen Cristiane Teixeira Vivaldini
Professor(es) Responsável(is)

Integrantes do Grupo

Daniel Amaral Brigante RA: 769867
Otavio de Paiva Pinheiro Neto RA: 769664

23 de março de 2022

Sumário

1. Introdução

O presente trabalho iniciou-se na busca por um "path finding"que fosse possível implementar na problemática apresentada e que fosse eficiente. Assim, o algoritmo escolhido foi o clássico "A*"por se apresentar uns dos mais eficientes entre os pesquisados e também um dos mais famosos, tornando a pesquisa mais simples e rápida caso haja algum problema na implementação. Segue uma breve explicação do algoritmo utilizado e quanto a implementação.

1.1. O algoritmo A*

Amplamente utilizado, o algoritmo A* baseia-se na propagação de pontos a partir da origem até que um destes pontos encontre o destino estipulado, a partir daí são elaborados *Checkpoints* que formem o caminho até a origem. Este caminho é otimizado pelo cálculo da heurística, ou seja, a cada ponto são verificadas a proximidade dos vizinhos com o destino e o custo até chegar a nesse vizinho e aí o de melhor heurística é escolhido como próximo *Checkpoint*.

1.2. Correção de bugs

Observou-se que o robô SMB estava "escorregando"no mapa requisitado "testWorld.world". Para corrigir esse problema foi utilizada a física do mapa "big_map_summer_school.world"utilizado em aulas anteriores, segue alteração:

```
<physics name='default_physics' default='0' type='ode'>
        <max_step_size>0.001</max_step_size>
        <real_time_factor>1</real_time_factor>
        <real_time_update_rate>1000</real_time_update_rate>
</physics>
```

Outra correção se deu no parâmetro "odom0"do arquivo " /workspace/src/smb_common/ /smb_control/config/localization.yaml"que chamava "smb_velocity_controller/odom"para chamar "odom". Dessa forma, consegui-se corrigir a divergência de localização que havia entre o mapa gerado e o mundo simulado conforme se navegava por ele.

2. Desenvolvimento

2.1. Mapeamento

Inicialmente, o mapa foi gerado no GAZEBO com os mesmos passos da aula 7: usando o pacote *teleop* para controlar o robô e o pacote *gmapping* para obter o mapa a partir do laser. Ao fim do processo, obtém-se a imagem do mapa no formato *.pgm* e um arquivo *.yaml* com os metadados (ponto de origem, resolução etc.).

2.2. Tratamento do Mapa

O mapa gerado foi dividido em 2, um que foi tratado para ficar no RVIZ e outro para o algoritmo A*:

O mapa utilizado no RVIZ passou apenas por um tratamento de *Trash Holding* para melhorar aspectos visuais durante a execução da simulação.

Já o mapa do A* passou é o mapa do RVIZ com a retirada das bordas e um redimensionamento. Com o *Trash Holding* conseguimos polarizar o mapa em apenas duas cores, ou preto ou branco, que correspondem a valores distintos, 0 e 255, respectivamente, daí se consegue facilmente diferenciar o que é obstáculo do que é caminho a percorrer. Com a retirada das bordas, reduzimos processamento desnecessário, uma vez que o robô não tem como passar pelas paredes do labirinto. Com o redimensionamento foi possível reduzir a quantidade de *pixels* com alguma perda de resolução, mas ainda com o suficiente para o cálculo da rota.

Uma vez que os *pixels* da imagem tipo *.pgm* ficam codificados em forma de membros de uma matriz e a reduzimos de uma imagem 4000X4000 para uma de 200X200, fica óbvia a redução de processamento e o ganho de eficiência do algoritmo que utilizará essa imagem.

2.3. A*

O algoritmo A* foi implementado como um ROSNode, com base no código A* do repositório *PythonRobotics*(1), modificando-o para permitir a leitura e interpretação do arquivo *.pgm* do mapa e publicar a saída do algoritmo em um tópico da rede ROS. Foi usada a linguagem *Python* com o módulo *rospy*.

O algoritmo utilizado exige alguns parâmetros para poder calcular a rota até o destino, que são:

1. Origem: As coordenadas de início do robô.

2. Destino: As coordenadas de fim do robô.

3. Mapa: As paredes são obtidas a partir do mapa gerado previamente. Para otimizar o processamento do mapa e obtenção dos obstáculos, o mapa foi alterado retirando-se a parte de fora labirinto[1], reduzindo suas dimensões e aplicando *Trash Holding*.

4. Nome do mapa: Deve-se definir qual mapa será utilizado para o cálculo da rota

5. Animação: Determina-se se o algoritmo deve ou não plotar em gráfico o mapa todos os pontos utilizados para o cálculo A* e o caminho obtido da origem ao destino previamente.

Com excessão do mapa, que exige um arquivo *.pgm* externo, todos esses parâmetros são setados no arquivo de configuração (*.yaml*).

Após seus processos internos são publicados os *checkpoints* no tópico de mesmo nome dentro do ROS, tais pontos são a entrada do Controlador.

2.4. Controlador

Além dos pontos obtidos pelo Algoritmo A* citados anteriormente, temos a entrada da AMCL e dos laseres. As entradas do AMCL e do A* são utilizadas no planejamento global da rota e os laseres, no planejamento local. O controlador foi implementado no ROSNode *smb highlevel controller* com a linguagem C++ e a biblioteca *roscpp*.

2.4.1. Planejamento Global

A AMCL realiza uma fusão de dados entre a odometria e o laser para se localizar no mapa fornecido ao programa pelo /map_server e, assim, o robô consegue saber onde ele se encontra. Já com as leituras dos *checkpoints* (enviados pelo A*) ele sabe onde deve ir.

Por conseguinte, a partir do momento em que se sabe onde está e para onde se vai é possível calcular a direção da origem ao destino e a seguir. A partir do vetor diferença entre a posição atual do robô e a posição do próximo *checkpoint*, um controlador proporcional determina a velocidade angular e linear, com base no ângulo do vetor diferença.

[1]E este requisito se torna fundamental, uma vez que o destino e origem do trabalho são setados desconsiderando tais "excessos"gerados na confecção do mapa a partir do *Gazebo*.

2.4.2. Planejamento Local

No caso deste trabalho, pretende-se regular uma margem de segurança entre o robô e os obstáculos, para tanto, foi elaborado este planejador. Quando um laser detecta proximidade com algo menor que a distância pré-definida em sua "frente", o robô reduz sua velocidade linear e desvia de forma a se alinhar paralelamente a esse ponto, voltando a seguir o planejador global quando esse ponto sai do alcance do controlador local. Isso é feito por meio de um controlador proporcional que utiliza como sinal de erro o produto escalar entre o vetor de orientação do robô e o vetor posição do obstáculo relativo ao robô (fornecido pelo laser).

3. Conclusão

Após uma cansativa otimização de parâmetros foi conseguido uma boa otimização do planejamento considerando-se o que havia no início do presente, tanto em questão de tempo para se completar o caminho, como o próprio andar pelo caminho em si, não passando a completar, mas evitando quaisquer colisões, perdas de odometria e ganhando velocidade quando possível.

Devido a alta generalização desenvolvida, pode-se afirmar, com certa tranquilidade, que em outro "world"do qual se tenha mapa e os outros requisitos já citados será calculada rota e seguida pelo robô. Basta se atentar quanto a possibilidade de se atingir o objetivo sem colidir ou atravessar alguma parede ou obstáculo.

Obs.: Em casos de caminhos muito estreitos deve-se realizar uma redução nas margens de segurança propostas, tendo em vista considerável aumento no tempo e processamento do projeto.

Repositório do Projeto

https://github.com/danielb-28/RMA

Referências

1 ATSUSHISAKAI. **PythonRobotics**. Disponível em: https://github.com/AtsushiSakai/PythonRobotics. Accesso em: 18 de março de 2022.