Trabalho Prático 2 - Robótica Móvel (DCC042)
Mapeamento

Hugo Araújo de Sousa

Departamento de Ciência da Computação
Instituto de Ciências Exatas
Universidade Federal de Minas Gerais
2º Semestre de 2018

hugosousa@dcc.ufmg.br

1. Introdução

No Trabalho Prático 1 da disciplina de Robótica Móvel, foi explorada a subárea de planejamento de caminhos para robôs móveis. Em particular, dois algoritmos foram implementados: *Bug* 2 e Campos Potenciais. Ambos os algoritmos são reativos, isto é, não necessitam de qualquer informação do ambiente além das leituras obtidas através de seus sensores (laser, sonar, etc...). Enquanto esse esquema é útil, pois permite encontrar caminhos mesmo em ambientes desconhecidos, os caminhos gerados tendem a não serem ótimos (em termos de seu comprimento), o que pode ser um grande problema para robôs com pouca energia para executar suas tarefas (o tempo é um fator crítico). Nesse contexto, o uso de mapas do ambiente pode facilitar a tarefa de navegação do robô, uma vez que, nesse caso, ele terá mais informações e tenderá a planejar caminhos mais curtos. Neste trabalho, o problema de mapeamento de ambientes é abordado através da implementação do algoritmo *Occupancy Grid*.

O algoritmo *Occupancy Grid* [Elfes 1989] funciona a partir da discretização do ambiente em que o robô se encontra em uma grade de células. Cada célula dessa grade representa uma área do ambiente e, para cada uma, armazena-se uma probabilidade que representa a chance da área no ambiente equivalente àquela célula estar ocupada (com um obstáculo). A atualização das probabilidades de ocupação de cada célula se dá a partir das leituras que o robô faz do ambiente através de seus sensores e de seu movimento ao longo do próprio ambiente. Um exemplo de mapa construído com essa técnica é mostrado na Figura 1.

2. Implementação

Assim como o primeiro trabalho prático da disciplina, este trabalho foi implementado utilizando-se o *framework* ROS (*Robot Operating System*) [Quigley et al. 2009, Yoonseok Pyo 2017] e a linguagem de programação Python 2.7.15 [1].

O primeiro passo na implementação foi a organização e reaproveitamento do código do trabalho prático 1. Para isso, foi criada uma classe `Robot` que implementava inicialmente somente funcionalidades de controle do robô e planejamento de caminhos da posição atual do robô até uma coordenada (utilizando o algoritmo *Bug 2*). Como a especificação do trabalho não impôs restrições em relação ao equipamento e tipo do robô utilizado, foi escolhido trabalhar com um robô holonômico com sensor *laser* de 180 graus.

[1] https://docs.python.org/2/index.html

Figura 1. Exemplo de mapa gerado através da técnica *Occupancy Grid*. Partes brancas indicam áreas livres no ambiente, enquanto partes cinzas indicam áreas desconhecidas e partes pretas indicam obstáculos.

Além disso, como o algoritmo *Occupancy Grid* necessita da posição exata do robô no ambiente (odometria perfeita), foi utilizado o tópico ROS `/base_pose_ground_truth`.

Para facilitar a implementação, o trabalho foi dividido em etapas, que são detalhadas nas subseções seguintes.

2.1. Modelando o mapa em uma grade de células

Como mencionado, o algoritmo *Occupancy Grid* representa o ambiente no qual o robô está inserido como uma grade de células (de tamanhos iguais). Dessa forma, foi necessário criar uma representação que seguisse essa descrição. A escolha foi criar uma classe `Grid` que representa o mapa da forma descrita e fornece funcionalidades básicas para sua manipulação. A grande de células foi implementada como um *array numpy*[2] bidimensional. Assim, é necessário, ao criar uma instância da classe `Grid`, especificar a altura e largura do mapa (em número de células) e o tamanho de cada célula (em metros), chamado aqui de resolução. Cada célula da grade é inicializada com uma probabilidade de estar ocupada $p = 0.5$, que representa o estado desconhecido (nem ocupada nem livre).

2.2. Identificando uma célula através de coordenadas no mundo

Uma vez que temos uma estrutura para representar a grade de células que modela o ambiente real do robô, algumas operações precisam ser implementadas para fornecer a interface entre as duas representações. A principal dessas operações é implementada na função `Grid.position_to_index`, que retorna o índice da célula na grade correspondente a uma coordenada (x, y) no ambiente.

A Figura 2 mostra a modelagem de um mapa em uma grade de células. Como vemos, em termos geométricos, os pontos do ambiente correspondem a uma grade onde

[2]https://docs.scipy.org/doc/numpy-1.15.1/reference/

as linhas de índice menor estão mais abaixo na grade e as colunas de índice menor estão à esquerda. Esse esquema é idêntico à estrutura de um *array numpy* bidimensional, em relação às colunas. Já para as linhas, é invertido, uma vez que as linhas de índice menor aparecem mais acima no *array* (quando impresso na tela). Dessa forma, é necessário espelhar a grade em relação ao eixo x quando precisamos que a grade tenha a mesma orientação que o ambiente.

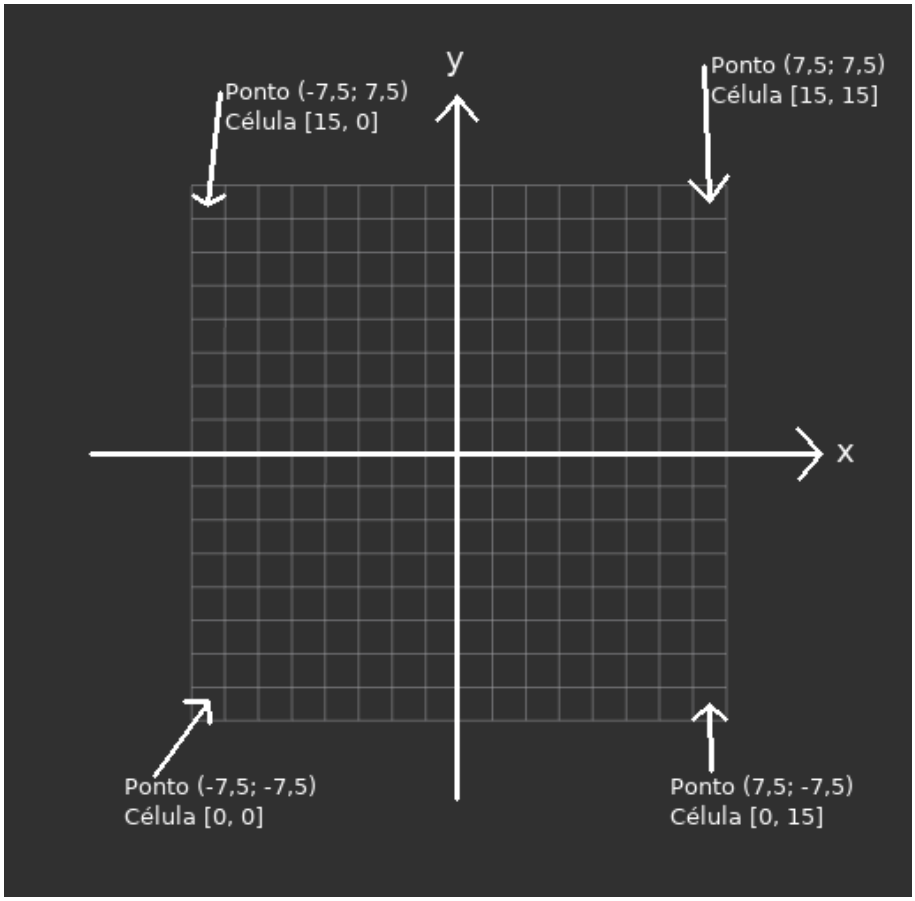


Figura 2. Exemplo de modelagem de um mapa 16m x 16m e a correspondência entre alguns pontos no mapa e os respectivos índices de célula na grade *numpy*.

2.3. Convertendo a grade em uma mensagem *Occupancy Grid*

Como especificado na documentação do ROS [3], uma grade de células que mapeia um ambiente pode ser transmitida por tópicos através de mensagens do tipo `OccupancyGrid`, do módulo `nav_msgs.msg`. O método `Grid.get_occupancy_msg` foi criado para realizar a conversão da estrutura de grade implementada na classe *Grid* para uma mensagem `OccupancyGrid`. Essas mensagens possuem um cabeçalho que informa o ID do *frame* do mapa, a resolução das células, a altura e largura do mapa, o ponto que representa sua origem e sua orientação. Realizando essas conversões e publicando as mensagens geradas, é possível visualizar o mapa sendo gerado em tempo real, à medida que o robô explora o ambiente, no programa Rviz [4].

[3] http://docs.ros.org/jade/api/nav_msgs/html/msg/OccupancyGrid.html
[4] http://wiki.ros.org/rviz

2.4. Identificando células ocupadas e livres

A atualização das células da grade que representa o ambiente ocorre após cada leitura do sensor *laser* do robô. Para cada um dos feixes de *laser* do sensor, é preciso identificar quais células da grade estão em seu campo de alcance. Isto é, quais células estão ao final do feixe (representando células ocupadas) e quais células estão ao longo do feixe de luz, porém antes do ponto de toque entre o feixe e o obstáculo observado (células livres). A Figura 3 ilustra esse esquema.

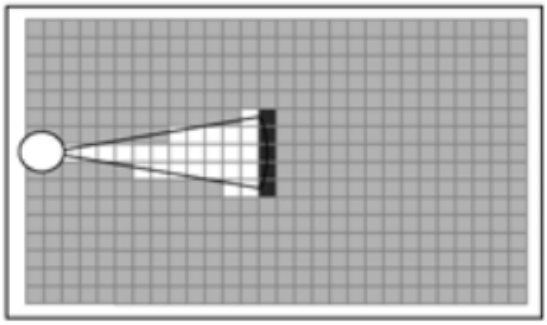


Figura 3. Exemplo de feixe de laser passando por células livres (em branco) e encontrando células ocupadas por obstáculos (em preto) [Thrun 2003].

Uma vez que a grade de células foi implementada como um *array numpy*, podemos realizar a identificação de células livres e ocupadas de forma sucinta e eficiente, através de máscaras de *booleanos*, que permitem atualizar somente parte da grade de probabilidades (as partes livres ou ocupadas, ignorando-se aquelas fora do alcance do laser), com base no cumprimento de condições estabelecidas. Esse processo foi implementado no método `Robot.apply_measurements_to_grid` que identifica as células livres e ocupadas (de acordo com a leitura do *laser*) e atualiza suas probabilidades como descrito na subseção 2.5.

A identificação de células livres e ocupadas necessita inicialmente obter, para cada célula, a distância de seu centro de massa até o robô (obtida através dos métodos `Grid.center_of_mass_from_index` e `Robot.get_dist_to_goal`) e o ângulo entre cada centro de massa e o robô. Caso **(i)** o ângulo entre o robô e o centro de massa de uma célula seja menor que metade da largura do feixe de *laser* e **(ii)** a distância entre o centro de massa da célula e o robô seja menor que a distância informada pelo feixe de *laser* subtraída da metade da suposta grossura dos obstáculos (um parâmetro do algoritmo), essa célula é considerada livre. Para identificar células ocupadas, a condição **(i)** também deve ser satisfeita, mas, além disso, **(iii)** o módulo da diferença entre a distância informada pelo *laser* e a distância entre o centro de massa da célula e o robô deve ser menor ou igual à metade da suposta grossura dos obstáculos. Essas condições foram adaptadas às estruturas do *array numpy*, a partir da descrição do algoritmo *Occupancy Grid* vista em [Thrun 2002].

2.5. Atualização de probabilidades

A partir da identificação de células livres e ocupadas em relação a cada feixe de *laser*, para cada leitura feita pelo robô, devemos atualizar a probabilidade de cada célula estar

ocupada. Para isso, o algoritmo utiliza a regra de Bayes. Entretanto, para tornar os cálculos mais eficientes, é preciso utilizar a representação *Log-Odds* das probabilidades, o que permite trabalhar apenas com somas, ao invés de multiplicações e divisões. Dado que:

$$Odd(X) = \frac{P(X)}{1 - P(X)}$$

a representação *Log-Odds* simplesmente retorna o logaritmo de *Odd(X)*. Dessa forma, escolhendo um limiar *p(occ)* que define um valor de probabilidade a partir do qual uma célula é considerada ocupada, podemos calcular:

$$l_{occ} = log(Odd(p(occ)))$$
$$l_{free} = log(Odd(1 - p(occ)))$$

Assim, inicialmente o valor $log(Odd(0.5))$ é atribuído a todas as células e, a cada leitura de *laser*, células livres são incrementadas ao valor l_{free} e células ocupadas são incrementadas ao valor l_{occ}. Ao final do mapeamento, é possível obter as probabilidades de que cada célula esteja ocupada ($p(m_i)$) a partir de seus *Log-Odds* ($l(m_i)$), com a fórmula a seguir, implementada no método `Grid.get_prob_from_log_odds`.

$$p(m_i) = 1 - (\frac{1}{1 + exp(l(m_i))})$$

Esse esquema de atualização de probabilidades foi retirado principalmente do curso *Robotics: Estimation and Learning* da Universidade da Pensilvânia na plataforma Coursera[5].

2.6. Esquema de navegação

Para que o robô cubra todo o ambiente onde está inserido (situação ideal para obtenção de mapas completos), é importante desenvolver alguma estratégia de navegação para guiá-lo no mapa. Neste trabalho, foi implementada uma estratégia de navegação que se baseia na chamada *Frontier Based Exploration* [Topiwala et al. 2018, Gu and Xu , Yamauchi 1997], onde o robô se move em direção às regiões de fronteira, isto é, regiões que formam uma fronteira entre regiões já marcadas como livres regiões ainda desconhecidas. O esquema de navegação implementado neste trabalho segue os seguintes passos:

1. Cheque se o número de pontos de fronteira a serem explorados já foi atingido. Se sim, para.
2. Selecione uma célula aleatória na grade.
3. Realize uma busca *Breadth-First Search* a partir dessa célula, seguindo em direção a todos seus vizinhos diretos.
4. Caso uma célula livre seja encontrada, checar se algum de seus vizinhos apresenta probabilidade de estar ocupado desconhecida (por volta de 0.5). Se sim, a célula atual é uma célula de fronteira.

[5]https://pt.coursera.org/lecture/robotics-learning/3-2-2-log-odd-update-Uh32O

5. Envia o robô na direção do centro de massa da célula de fronteira encontrada.

Como a busca começa em uma célula aleatória, diferentes execuções do programa gerarão mapas diferentes. O método responsável pela obtenção de um novo alvo (fronteira) ao qual enviar o robô para exploração é o `Grid.get_random_frontier_cell`.

2.7. Salvando o mapa gerado em um arquivo

Ao fim da exploração do número de fronteiras aleatórias definidas como parâmetro no programa, o robô considera terminado o mapeamento do ambiente. Nesse ponto, é interessante gerar alguma saída que permita verificar visualmente o mapa gerado. O método `Grid.dump_pgm` é responsável por isso e escreve em um arquivo PGM[6] os dados do mapa gerado, ao fim da execução do programa. Caso o programa seja executado com o utilitário `rosrun`, o mapa será salvo no próprio diretório do projeto no *workspace*. Caso seja executado com `roslaunch`, o mapa será salvo no diretório `~/.ros` (sistemas Linux). Em ambos os casos, o nome do arquivo de mapa será `map.pgm`.

3. Execução e Estrutura do Projeto

O algoritmo foi implementado sob um pacote ROS de nome `tp2`. O programa Python está implementado no arquivo `tp2.py`, que recebe como parâmetros obrigatórios de linha de comando a altura e largura da grade (em células), o tamanho de cada célula em metros (resolução) e o número de fronteiras aleatórias a serem exploradas. Dessa forma, podemos executar o programa da seguinte forma (após a execução dos nodos `roscore` e `stage_ros`).

```
rosrun tp2 tp2.py <h> <w> <r> <f>
```

Onde `<h>` se refere à altura da grade, `<w>` se refere à largura da grade, `<r>` se refere à resolução das células e `<f>` se refere ao número de fronteiras aleatórias a serem exploradas. Além disso, para executar todos os nodos necessários de uma só vez e rodar o algoritmo, podemos usar o arquivo `launch` do ROS, da seguinte forma:

```
roslaunch tp2 tp2.launch height:=<h> width:=<w>
 resolution:=<r> frontiers:=<f> world_file:=<m>
```

Onde os parâmetros são os mesmos mostrados acima, com adição de `<m>` que se refere ao nome do arquivo de mapa utilizado pelo simulador.

A fim de garantir o funcionamento correto do projeto, é importante que a estrutura do pacote no *workspace* ROS seja similar à mostrada na Figura 4.

4. Testes

A fim de verificar a corretude da implementação do algoritmo e como ele se comporta em ambientes com tipos de obstáculos diferentes, foram executados vários testes. Os três principais são mostrados nessa seção, todos utilizando mapas diferentes. O primeiro mapa foi tirado dos mapas disponíveis para o trabalho prático 1, enquanto o segundo foi criado para esse trabalho e o terceiro foi adaptado de um dos mapas disponíveis para o primeiro trabalho da disciplina. Para todos os mapas testados, o mapa foi modelado como um

[6] http://netpbm.sourceforge.net/doc/pgm.html

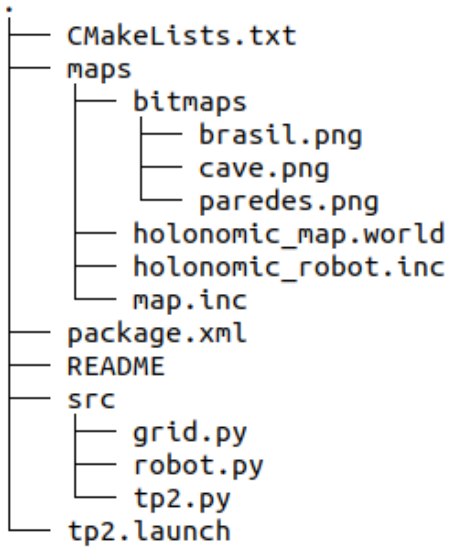
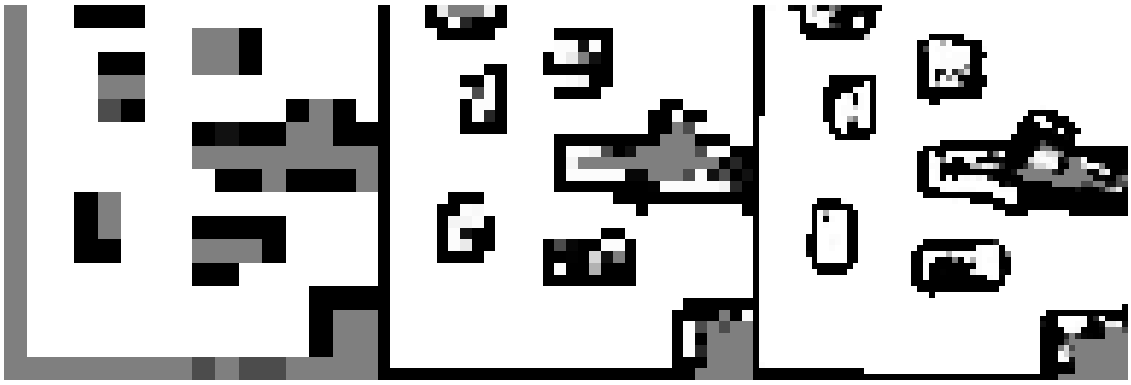


Figura 4. Árvore do diretório do pacote do projeto no *workspace* do ROS.

quadrado de dimensões 16m x 16m. Cada mapa foi testado 3 vezes, sendo os parâmetros de altura, largura e resolução configurados como **16, 16** e **1** inicialmente, com os dois primeiros multiplicados por 2 para cada nova execução em um mapa e o terceiro divido por 2. A descrição de cada um dos testes e os resultados encontrados são mostrados a seguir.

- **Teste 1: Mapa cave.png**. Cenário: muitos obstáculos de tamanho grande. Resultado mostrado na Figura 5



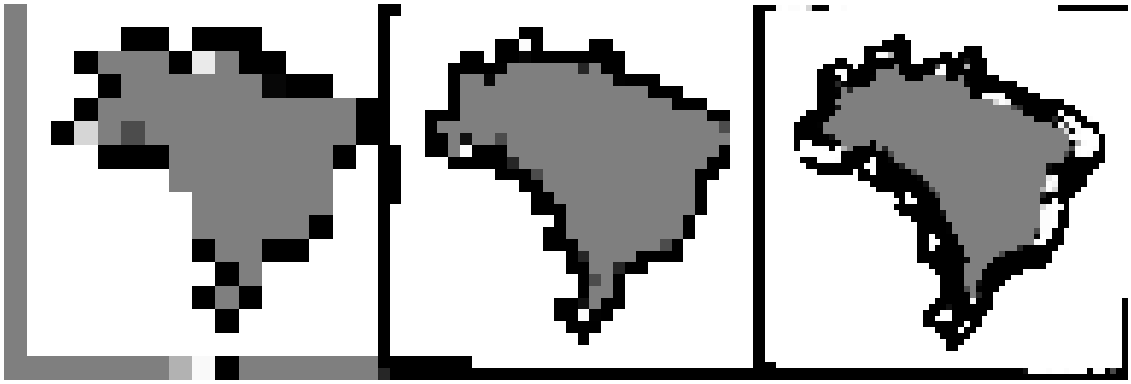
(a) 16x16 células. Resolução 1. (b) 32x32 células. Resolução 0.5. (c) 64x64 células. Resolução 0.25.

Figura 5. Execução do programa com o mapa **cave.png**.

Para esse teste, a intenção inicial era simplesmente verificar o funcionamento correto do algoritmo. Como podemos ver, já na imagem 5 (a), o contorno dos obstáculos pode ser verificado. Entretanto, aumentando o número de células e reduzindo seus tamanhos, o contorno dos obstáculos se torna cada vez mais similar ao mapa real. Com esse teste, um aspecto curioso foi observado: algumas áreas internas do obstáculos são marcadas como livres (em branco). Um palpite sobre esse comportamento é que, por vezes, o laser do robô durante a simulação

não é fiel e acaba passando por obstáculos.

- **Teste 2: brasil.png.** Cenário: obstáculo único, muito grande e de bordas muito irregulares. Resultado mostrado na Figura 6.

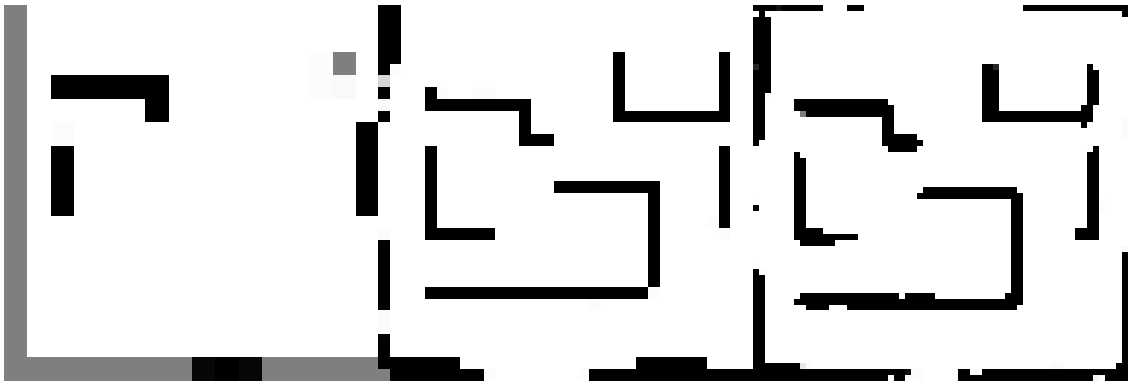


(a) 16x16 células. Resolução 1. (b) 32x32 células. Resolução 0.5. (c) 64x64 células. Resolução 0.25.

Figura 6. Execução do programa com o mapa **brasil.png**.

Nesse teste, o objetivo principal foi observar a precisão do mapeamento em termos do nível de detalhe do mapa obtido. Para isso, foi criado um mapa com um obstáculo central no formato do mapa do Brasil. Dessa forma, o contorno do obstáculo, visto de cima em uma visão 2D, tem um desenho irregular com curvas e traços abruptos. Verifica-se que quando menor o tamanho das células mais próximo do desenho real se torna o mapa obtido.

- **Teste 3: paredes.png.** Cenário: muitos obstáculos de espessura pequena. Resultado mostrado na Figura 7.



(a) 16x16 células. Resolução 1. (b) 32x32 células. Resolução 0.5. (c) 64x64 células. Resolução 0.25.

Figura 7. Execução do programa com o mapa **paredes.png**.

Nesse teste, o objetivo era verificar como o mapeamento se comportava com obstáculos de espessura muito fina, tais como os presentes no mapa **paredes.png**. No código, como mencionado, existe um parâmetro importante para a detecção de células livres e ocupadas em uma medição laser: a suposta espessura dos

obstáculos. Como a intenção aqui é justamente descobrir o mapa de um ambiente desconhecido, esse parâmetro foi definido com um valor de "chute", e não configurado como argumento do programa definido pelo usuário. Os mapas resultantes do teste mostram que, para células muito grandes (resolução 1), o algoritmo nem mesmo é capaz de identificar vários dos obstáculos. Isso se deve ao fato de que, em uma célula, mesmo que ela possua parte de um obstáculo, a sua área livre é maior, devido à espessura da parede obstáculo. Vemos que esse problema desaparece quando diminuímos o tamanho das células.

5. Conclusão

Neste trabalho foi implementado o algoritmo *Occupancy Grid* para mapeamento de ambientes por robôs móveis. O algoritmo teve como base o arcabouço de navegação e controle de robô holonômico desenvolvido no primeiro trabalho prático da disciplina Robótica Móvel.

De maneira geral, o desenvolvimento desse segundo trabalho se deu de forma mais tranquila do que o primeiro trabalho da disciplina, uma vez que já existia uma certa familiaridade com o *framework* ROS e os aspectos fundamentais de Robótica Móvel. A tarefa de maior dificuldade foi a de implementar os aspectos mais técnicos do próprio algoritmo, que utiliza fortemente vários conceitos da Teoria da Probabilidade. Além disso, também foi muito importante modelar bem a estrutura do mapa e sua correspondência ao ambiente do robô antes de começar o desenvolvimento do projeto.

Acredito que o aprendizado obtido tenha sido grande e complementar às aulas da disciplina sobre Robótica Probabilística, além do trabalho ter se mostrado um passo lógico após o trabalho prático 1. Os testes executados serviram para mostrar nuances no desempenho do algoritmo, assim como aspectos particulares do *framework* ROS.

Referências

Elfes, A. (1989). Using occupancy grids for mobile robot perception and navigation. *Computer*, (6):46–57.

Gu, T. and Xu, Z. Frontier based exploration for map building.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan.

Thrun, S. (2002). Probabilistic robotics. *Communications of the ACM*, 45(3):52–57.

Thrun, S. (2003). Learning occupancy grid maps with forward sensor models. *Autonomous robots*, 15(2):111–127.

Topiwala, A., Inani, P., and Kathpal, A. (2018). Frontier based exploration for autonomous robot. *arXiv preprint arXiv:1806.03581*.

Yamauchi, B. (1997). A frontier-based approach for autonomous exploration. In *Computational Intelligence in Robotics and Automation, 1997. CIRA'97., Proceedings., 1997 IEEE International Symposium on*, pages 146–151. IEEE.

Yoonseok Pyo, Hancheol Cho, L. J. D. L. (2017). *ROS Robot Programming (English)*. ROBOTIS.