Universidade Federal do Rio Grande do Sul
Instituto de Informática
INF01034 - Robótica II
Prof. Renan Maffei

Trabalho Prático 3 - Exploração usando A* e Campos Potenciais

Prazo: 04/06/2019

Considerações gerais

O trabalho deverá ser feito preferencialmente em DUPLA (senão INDIVIDUALMENTE).

A avaliação do trabalho será feita através da análise do funcionamento das implementações: primeiramente no simulador MobileSim, e posteriormente com o robô real. O teste no robô real só poderá ser feito quando a implementação estiver funcionando bem no simulador.

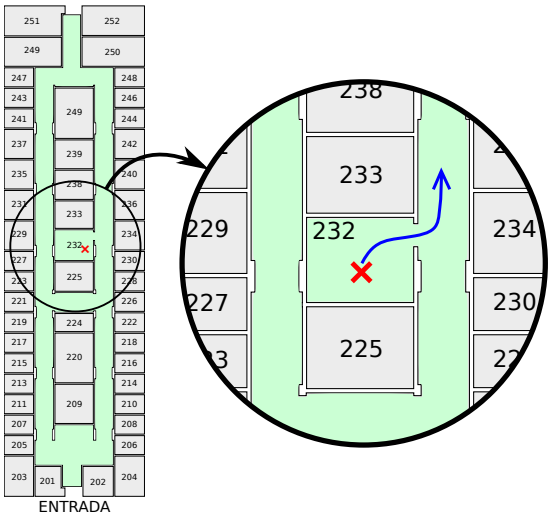

Figura 1: Durante o teste real, o robô deverá ser capaz de sair da sala de testes autonomamente. No teste simulado, o robô deverá explorar completamente o ambiente.

Além de apresentar o funcionamento da implementação daquilo que for pedido, entregue via Moodle um arquivo compactado com os códigos desenvolvidos.

Instruções sobre implementação do algoritmo de exploração no framework

Foi adicionada ao framework utilizado nos trabalhos da disciplina uma classe chamada `Planning` descrita no arquivo `Planning.cpp`. Esta classe é responsável por realizar continuamente a atualização do planejamento de caminhos usado na exploração do ambiente.

O framework agora possui três *threads*:

1. a *thread* principal que roda continuamente a função `run` da classe `Robot` (responsável pelo controle do robô e mapeamento do ambiente);
2. uma *thread* secundária que fica atualizando o desenho do grid e lendo a entrada do teclado;
3. uma nova *thread* que roda continuamente a função `run` da classe `Planning`, realizando a atualização do planejamento de caminhos para a exploração do mapa.

Como a thread de controle do robô modifica informações do grid que são utilizadas na atualização do campo potencial, um mutex é utilizado nas duas funções `run` para tratar o acesso à região crítica. Não há necessidade de modificar tais funções.

A versão atual do framework contém 5 modos de visualização do mapa, que podem ser alternados pressionando a tecla 'v'.

0. Mapa mostrando a classificação do ambiente, as fronteiras e o menor caminho escolhido *(default)*
1. Mapa mostrando o campo potencial local
2. Mapa com LOG-ODDS usando LASER
3. Mapa com HIMM usando LASER
4. Mapa usando SONAR

OBS: por enquanto todos os mapas estão em branco, pois vocês precisam colocar a função de mapeamento que desejam usar.

1. Implementação do método de limiarização da ocupação das células para classificá-las

Complete a função `updateCellsTypes` da classe `Planning`.

Essa primeira etapa do trabalho consiste em definir os tipos das células usadas no algoritmo de exploração, ou seja, definir a classificação de cada célula (`Cell* c`) entre:

- **não-exploradas**: `c->occType = UNEXPLORED` (todas as células começam com esse tipo)
- **livres**: `c->occType = FREE`
- **obstáculos**: `c->occType = OCCUPIED`

Você deve varrer todas as células ao redor do robô dentro do alcance máximo dos sensores (definido na classe `Planning` pela variável `maxUpdateRange`).
Para cada célula varrida é preciso analisar o valor de ocupação obtido por um método de mapeamento (pode ser tanto o HIMM quanto o log-odds usando laser - evite sonar). Defina limiares adequados para modificar a classificação de células originalmente UNEXPLORED para FREE ou OCCUPIED.
No momento em que uma célula deixa de ser UNEXPLORED, ela só pode alternar entre FREE ou OCCUPIED.

Dica: Use limiares diferentes para alternar de FREE para OCCUPIED (e vice-versa) para evitar que pequenas variações na ocupação em valores próximos do limiar causem oscilações na classificação.

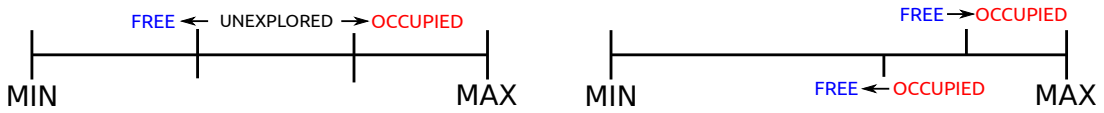

Figura 2: Escolha de limiares para alternar a classificação das células do grid.

OBS: Após a execução da limiarização é feito o crescimento dos obstáculos (através da função `expandObstacles` que já está pronta). Isso ajuda a evitar com que o robô chegue muito próximo de obstáculos. Essa função modifica a classificação de células FREE que sejam vizinhas a OCCUPIED para o tipo NEAROBSTACLE. Células desse tipo são visualizadas em cinza escuro no mapa.

OBS: Na sequência o programa detecta as fronteiras entre espaço livre e espaço desconhecido (através da função `detectFrontiers` que já está pronta). A função classifica as células de fronteira no campo `c->planType` como FRONTIER (para a célula central de uma fronteira) e MARKED_FRONTIER (para as demais células de fronteira). No final da função, é atualizado um vetor contendo todos os centros das fronteiras (`std::vector<Cell*> frontierCenters`)

2. Implementação do método para computar a função de heurística do A* em todas as células livres

Complete a função `computeHeuristic` da classe `Planning`. Atualize o campo `c->h` de todas as células livres (`c->occType = FREE`) e células objetivo, i.e. de centro de fronteira (`c->planType = FRONTIER`) do grid.

O valor de heurística usado deverá ser a menor distância para o objetivo mais próximo. Nas células de objetivo, o valor de `c->h` será obviamente 0. Nas demais células, é preciso determinar a distância euclidiana para a célula de objetivo mais próxima dentre todas em `std::vector<Cell*> frontierCenters`. Lembre-se que este valor é uma heuristica usada para acelerar a busca do menor caminho, ou seja, é simplesmente um valor hipotético da menor distância entre cada célula e o objetivo mais próximo (caso não houvesse obstáculos no mapa).

3. Implementação do método A* para computar o menor caminho a partir da posição do robô até a fronteira mais próxima

Complete a função `computeAStar` da classe `Planning`. Os valores utilizados no algoritmo A* associados a cada célula do grid estão nos campos `c->f`, `c->g` e `c->h` (todos do tipo double, inicializados com valor DBL_MAX) e o ponteiro `Cell* c->pi` (inicializado com NULL).

O algoritmo utiliza uma fila de prioridades de células, ordenadas pela chave f. Para facilitar o trabalho, é aconselhável usar a fila de prioridades declarada no início da função.

A busca do menor caminho deve ser propagada adicionando células vizinhas na fila até que se encontre uma célula de fronteira (i.e. até achar uma célula onde `c->planType == FRONTIER`). Esta célula deve ser setada como objetivo, i.e. fazer `goal=c`.

OBS: Após a execução da função, o menor caminho é gerado de trás pra frente a partir da célula `goal`, seguindo o ponteiro `c->pi` para o antecessor de cada célula até alcançar a posição do robô (em que `c->pi = NULL`). Isso está implementado na função `markPathCells`.

OBS: A seguir um objetivo local (próximo ao robô) é definido na função `findLocalGoal`. Este objetivo é a primeira célula no caminho encontrado pelo A* cuja distância para o robô é menor do que um dado threshold `localGoalRadius`. O `localGoal` será a única célula de potencial atrator dentro de uma janela local centrada no robô, sobre a qual será computado um campo potencial harmônico. A inicialização das condições de contorno do campo potencial está feita na função `initializePotentials`.

4. Implementação do método de atualização do campo potencial

Complete a função `iteratePotentials` da classe `Planning`. Para isso varra todas as células dentro de uma janela local ao redor do robô de raio `halfWindowSize`. No código, o potencial de uma célula (`Cell* c`) é dado por `c->pot`.

Note que cada chamada dessa função executa UMA ITERAÇÃO de atualização do campo potencial, ou seja, atualiza cada célula uma vez só. Porém, como essa função é chamada repetidamente dentro da função `run` e o tamanho da janela é pequeno, o campo potencial deverá convergir rapidamente.

No campo potencial local que deverá ser computado existe apenas uma célula com potencial atrator fixo, o localGoal. Esta célula pode ser verificada testando c->planType == LOCALGOAL. Já os obstáculos (c->occType == OCCUPIED) continuam tendo potencial repulsivo fixo. Todas as demais células (occtype FREE ou UNEXPLORED) devem ter seu potencial atualizado pela equação de Laplace usando o método de diferenças finitas (Gauss-Seidel).

$$p(\boldsymbol{c}_{i,j}) \leftarrow \frac{p(\boldsymbol{c}_{i,j+1}) + p(\boldsymbol{c}_{i,j-1}) + p(\boldsymbol{c}_{i+1,j}) + p(\boldsymbol{c}_{i-1,j})}{4} \tag{1}$$

5. Implementação do método de atualização do gradiente do campo potencial

Complete a função `updateGradient` da classe `Planning`.

Nesta etapa é preciso computar o vetor gradiente descendente ($-\nabla p$) das células FREE, através de uma varredura igual a que é feita no exercício anterior. Cada célula (`Cell* c`) do grid tem este vetor descrito pelas componentes horizontais e verticais: `c->dirX` e `c->dirY`

$$-\nabla p(\boldsymbol{c}_{i,j}) \leftarrow \left(\underbrace{-\frac{p(\boldsymbol{c}_{i+1,j}) - p(\boldsymbol{c}_{i-1,j})}{2}}_{dirX}, \underbrace{-\frac{p(\boldsymbol{c}_{i,j+1}) - p(\boldsymbol{c}_{i,j-1})}{2}}_{dirY} \right)$$

Lembre de normalizar o vetor gradiente pois para a navegação só nós interessa saber a direção do gradiente, não sua intensidade. Ou seja, divida as componentes horizontais e verticais pela norma do gradiente $|\nabla p|$,

$$|\nabla p| \leftarrow \sqrt{(dirX)^2 + (dirY)^2}$$

Por fim, não é preciso computar o gradiente nas células diferentes de FREE pois elas não são navegáveis, então basta colocar `c->dirX` e `c->dirY` iguais a 0.

Para visualizar o gradiente, aperte a tecla 'f'.

OBS: Nesta nova versão do framework há um novo modo de navegação POTFIELD que pode ser ativado pressionando a tecla '5' [1]. Nesse modo cabe ao robô seguir o gradiente descente do campo potencial referente a célula que ele se encontra sobre. Isso é feito na função `followPotentialField` da classe `Robot` usando controle PID.

[1] As teclas de 1 a 4 ativam os modos de navegação implementados no primeiro trabalho.