Instituto Tecnológico de Aeronáutica – ITA
Inteligência Artificial para Robótica Móvel – CT-213
Professor: Marcos Ricardo Omena de Albuquerque Maximo

Laboratório 2 – Busca Informada

Observações:

- Tive dificuldades para fazer o numpy funcionar no Windows com o Python 3.7. Assim, se não conseguir também, recomendo criar um novo environment no Anaconda usando Python 3.6.
- Além dos pacotes que havia instalado antes, também precisa do “matplotlib” para esse lab.

1. Introdução

Nesse laboratório, seu objetivo é implementar um planejador de caminho para um robô móvel. Imagine um caso com o mostrado na Figura 1(a), em que um robô está num determinado ambiente com obstáculos e quer encontrar um caminho até um objetivo. Como mostrado em aula, pode-se usar busca em grafos para determinar este caminho, conforme ilustra a Figura 1(b). Neste laboratório, você implementará os algoritmos Dijkstra, Greedy Search e A* para realizar resolver esse problema de busca em grafos.

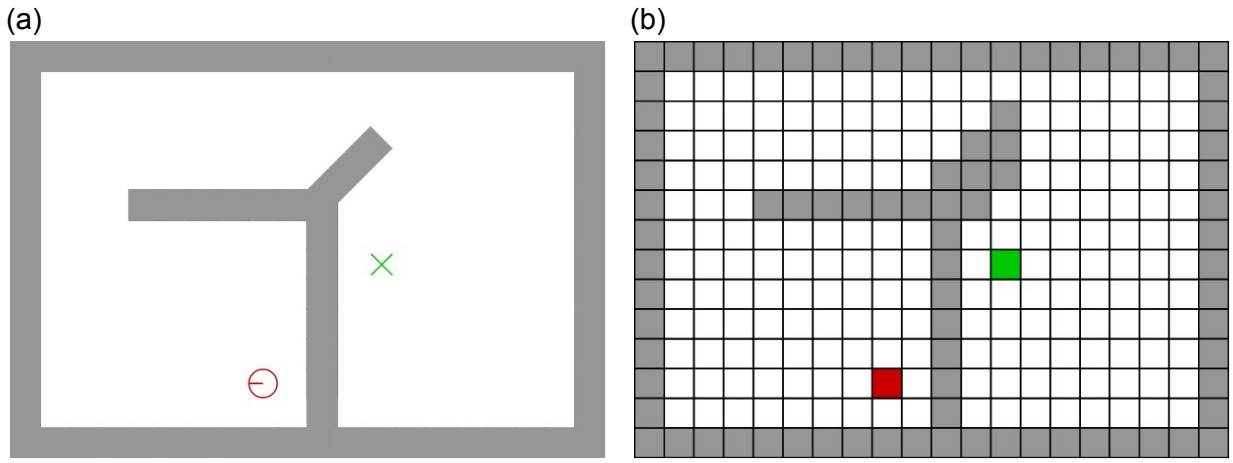

Figura 1: planejamento de caminho de robô móvel usando busca em grafos.

2. Descrição do Ambiente

No código base fornecido, o ambiente já é um grid, de modo que não será necessário converter um mapa contínuo para um mapa discreto (*grid*). Considere que o *grid* é 8-conectado

(ver Figura 2), i.e. para cada célula, há 8 vizinhos. Além disso, considere que o custo de movimento no *grid* é:

$$C(c_i, c_j) = f * (C(c_i) + C(c_j))/2 ,$$

em que $C(c_i, c_j)$ é o custo para movimento entre as células c_i e c_j; $C(c_i)$ e $C(c_j)$ são os custos associados às células c_i e c_j; e f é um fator multiplicativo que vale $\sqrt{2}$ se o movimento for em diagonal e 1 se for horizontal ou vertical.

O mapa considerado nesse laboratório é o mostrado na Figura 3, em que os obstáculos estão representado em roxo escuro, as células "livres" estão exibidos em verde e regiões próximas ao obstáculos estão pintadas de amarelo. A região verde tem custo de célula 1, enquanto as amarelas tem custo de célula 2 a fim de induzir os algoritmos de busca a encontrarem caminhos mais seguros. Os obstáculos representam regiões completamente proibidas. Ademais, na Figura 3, o estado inicial está indicado com uma estrela amarela, o objetivo com uma cruz vermelha e o caminho encontrado pelo algoritmo Dijkstra com linhas azuis.

Finalmente, a Figura 3 destaca uma confusão comum: a representação comum para eixos coordenados é usar (x, y), em que x é o eixo horizontal e y é o vertical, porém a representação usual para *grid* (proveniente de matrizes) é usar (i, j), em que i indica a linha e j indica a coluna. Cuidado: há uma inversão de ordem entre (x, y) e (i, j) que costuma gerar confusão. O código fornecido trabalha na convenção (i, j). Destaca-se ainda que os termos *width* (largura) e *height* (altura), que são termos referentes à imagem do tabuleiro, se referem ao número de colunas e de linhas, respectivamente.

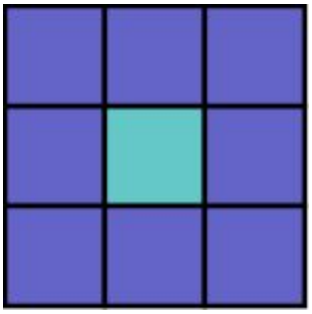

Figura 2: representação de 8-conectado.

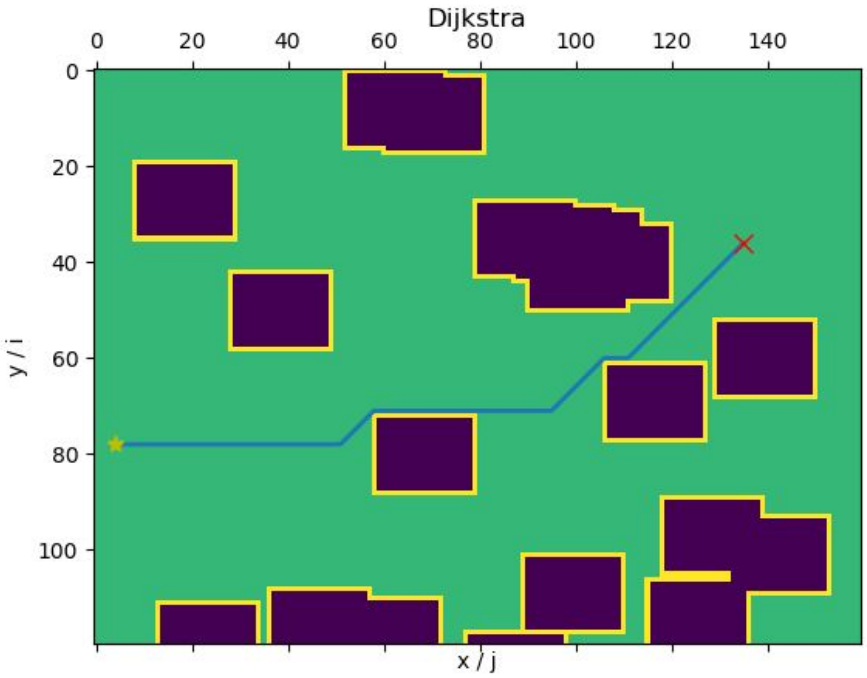

Figura 3: exemplo de caminho ótimo no ambiente.

3. Código Base

Junto com esse roteiro, foi entregue um código base. Esse código base contém boa parte da implementação do laboratório. A criação do mapa e estruturas auxiliares já estão implementadas. A ideia é que você apenas implemente os algoritmos Dijkstra, Greedy Search e A*. As seguintes classes foram entregues:

- `CostMap`: representa o mapa de custo. O método `get_edge_cost()` recebe posições (i, j) dos dois vértices de uma aresta e calcula o custo da aresta.
- `NodeGrid`: representa o grafo onde o planejamento será realizado. O método `get_successors()` é útil para obter todos os sucessores (8-conectado) de um determinado nó.
- `Node`: uma célula do *grid*. A célula possui variáveis interessantes para facilitar a implementação: a posição (i, j) da célula no *grid*, os valores de $f(n)$ e $g(n)$ usados na busca em grafo, o pai do nó na árvore de busca e uma variável booleana para indicar se o nó já está "fechado" (i.e. já está com sua distância definida). Finalmente, o método `distance_to()` calcula a distância euclidiana entre esse nó e uma outra posição (i, j) no *grid* para uso como heurística.
- `PathPlanner`: classe que efetivamente executa o planejamento de caminho. O método `construct_path()` é semelhante ao método mostrado em sala e extrai um caminho do grafo, dado que os membros `parent` dos nós da árvore de busca tenham sido preenchidos corretamente.

Destaca-se que em alguns momentos da implementação considerada, usa-se uma tupla (de Python) (i, j) para representar uma posição do tabuleiro. O script principal `main.py` executa planejamentos de caminho usando um dos 3 algoritmos implementados, traça gráficos mostrando os caminhos encontrados e calcula estatísticas (média e desvio padrão) de tempo computacional e de custo do caminho.

4. Tarefas

Implemente os métodos `dijkstra()`, `greedy()` e `a_star()` da classe `PathPlanner`. Para ajudar você a debugar sua implementação, o código vem configurado para mostrar uma figura (semelhante à Figura 3) e salvá-la no seu computador em formato .png. Além disso, por questões de reprodutibilidade, não altere a semente de geração de números aleatórios configurada no código. Inclua no seu relatório figuras mostrando os caminhos encontrados pelos três algoritmos para o primeiro problema gerado pelo código.

Quando estiver certo da sua implementação, compare os algoritmos usando um Monte Carlo (basicamente, trata-se de repetir o algoritmo várias vezes, com célula inicial e objetivo aleatórias, de modo a testar o algoritmo em várias situações diferentes) com 100 iterações. Para execução do Monte Carlo, recomenda-se desativar exibição e gravação das figuras nesse caso. Por fim, inclua no seu relatório os resultados do Monte Carlo através de uma tabela semelhante à Tabela 1.

Algoritmo	**Tempo computacional (s)**		**Custo do caminho**	
	Média	**Desvio padrão**	**Média**	**Desvio padrão**
Dijkstra				
Greedy Search				
A*				

Tabela 1: tabela de comparação entre os algoritmos de planejamento de caminho.

Lembre-se de usar uma fila de prioridades para que os algoritmos fiquem eficientes. Uma recomendação de implementação de fila de prioridades é a `heapq` do Python. Para usá-la:

- Criação: `pq = []`.
- Inserção: `heapq.heappush(pq, (node.f, node))`.
- Extração: `f, node = heapq.heappop(pq)`.

No caso dessa implementação, não é possível atualizar um nó já inserido, então um nó pode ser inserido mais de uma vez na fila de prioridades. Para evitar que isso gere problemas, use o atributo `node.closed` que indica se um nó já está “fechado” (com distância definida). Observação: você é o responsável por gerenciar o atributo `node.closed`, inclusive por setá-lo

como `True` no momento correto. Caso queira, fique livre para usar outra implementação de fila de prioridades.

5. Entrega

A entrega consiste do código e de um relatório, submetida através do Google Classroom. Modificações nos arquivos do código base são permitidas, desde que o nome e a interface dos scripts “main” não sejam alterados. A princípio, não há limitação de número de páginas para o relatório, mas pede-se que seja sucinto. O relatório deve conter:

- Breve descrição em alto nível da sua implementação.
- Figuras que comprovem o funcionamento do seu código.

Por limitações do Google Classroom (e por motivo de facilitar a automatização da correção), entregue seu laboratório com todos os arquivos num único arquivo **.zip** (**não** utilize outras tecnologias de compactação de arquivos) com o seguinte padrão de nome: “**<login_email_google_education>_labX.zip**”. Por exemplo, no meu caso, meu login Google Education é **marcos.maximo**, logo eu entregaria o lab 2 como “**marcos.maximo_lab2.zip**”. **Não** crie subpastas para os arquivos da sua entrega, **deixe todos os arquivos na “raiz” do .zip**. Os relatórios devem ser entregues em formato **.pdf**.