**Apresentação IA (16,17,18,19,20,21,22)**

**17 -** A heurística da distância de Manhattan é usada em algoritmos de busca como o A Estrela e o Greedy Search para estimar o custo de atingir um objetivo a partir de um ponto inicial. Este cálculo é especialmente útil em ambientes representados por grades onde os movimentos permitidos ocorrem apenas na horizontal ou vertical. A fórmula da distância de Manhattan é a soma das diferenças absolutas das coordenadas x e y entre o ponto inicial e o objetivo. A principal vantagem desta heurística esta na sua simplicidade e eficiência.

**18/19 -** Este tipo de procura expande para o nodo que parece estar mais perto da solução utilizando para isso uma heurística. Este tipo tem como vantagens um tempo de procura reduzido se a função heurística for boa, no entanto a solução obtida pode não ser a solução ótima e ainda ocupa muito espaço em memória.

Aqui temos a nossa função, então primeiro criamos um no inicial com a posição de partida **(start\_node = Node(start))**, definimos ainda os nos objetivos, ou seja, as áreas a que queremos chegar, verificamos se o no inicial e os nos objetivo existem no grafo. No loop principal **(while not queue.empty():),** pegamos no nó com menor valor heurístico da fila, senão foi visitado, marcamos como visitado e adiciona-o ao caminho explorado, **if current in end\_nodes:** se chegou ao objetivo, reconstrói o caminho e retorna-o, **for neighbor in self.g.neighbors(current):**, senao para cada vizinho não visitado, define o nó atual como pai do mesmo, calcula a prioridade usando a distancia de Manhattan(heurística) ate ao objetivo e adiciona a fila de espera

**20/21 -** Este tipo de procura evita expandir caminhos que são dispendiosos, combinando para isso o algoritmo de procura gulosa com o algoritmo de procura uniforme

A vantagem deste tipo de procura será a obtenção da solução ótima se a sua heurística for admissível, no entanto, este também ocupa muito espaço em memória.

Tal como na Gulosa, criamos um no inicial com a posição de partida **(start\_node = Node(start))**, definimos ainda os nos objetivos, ou seja, as áreas a que queremos chegar, verificamos se o no inicial e os nos objetivo existem no grafo. Depois teremos **g\_score = {start\_node: 0}** que será o custo real do início ao no atual e o **f\_score** que será a nossa função ou seja o **g\_score** mais a **heurística**, depois temos, **open\_set**: fila de prioridade com nós a serem explorados, **closed\_set:** conjunto de nós já explorados, **parent:** dicionário para reconstruir o caminho, **path\_traversed**: lista de nós visitados na ordem. Agora no loop principal **while not open\_set.empty():** pegamos no nó com menor **f\_score** da fila verificamos se já foi explorado, senão foi adicionamos ao conjunto de explorados, **if current in end\_nodes:** se chegou ao objetivo reconstrói o caminho e retorna-o, senão, **for neighbor in self.g.neighbors(current):** para cada vizinho não explorado calculamos o **tentative\_g\_score**, **if neighbor not in g\_score or tentative\_g\_score < g\_score[neighbor]:** se for a primeira vez que visitamos este nó ou encontramos um caminho mais curto ate este nó atualizamos o **pai do vizinho para ser o nó atua**l, atualizamos o **g\_score** e calculamos o novo **f\_score** por fim adicionamos a fila de prioridade

**22 -** Concluindo, acreditamos que conseguimos desenvolver um programa que permita a utilização de algoritmos de procura para explorar caminhos possíveis e atingir uma solução, tendo sido descrita todas as vantagens e desvantagens de cada estratégia.