Отчёт по дз 2.

1. **Дейкстра**

def dijkstra(map\_array, start\_cell, goal\_cell):  
 distance\_dict = {}  
 for row in range(MAP\_HEIGHT):  
 for column in range(MAP\_WIDTH):  
 distance\_dict[(column, row)] = float('inf')  
   
 distance\_dict[start\_cell] = 0  
 previous\_dict = {}  
 queue = [(0, start\_cell)]  
  
 while queue:  
   
 current\_distance, current\_cell = heapq.heappop(queue)  
  
 if current\_cell == goal\_cell:  
 break  
  
   
 for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:  
 neighbor\_cell = (current\_cell[0] + dx, current\_cell[1] + dy)  
 if neighbor\_cell[0] < 0 or neighbor\_cell[0] >= MAP\_WIDTH or neighbor\_cell[1] < 0 or neighbor\_cell[1] >= MAP\_HEIGHT:  
 continue  
   
 if map\_array[neighbor\_cell[1]][neighbor\_cell[0]][2]:  
 continue  
   
 neighbor\_distance = current\_distance + distance(current\_cell, neighbor\_cell)  
   
 if neighbor\_distance < distance\_dict[neighbor\_cell]:  
 distance\_dict[neighbor\_cell] = neighbor\_distance  
 previous\_dict[neighbor\_cell] = current\_cell  
 heapq.heappush(queue, (neighbor\_distance, neighbor\_cell))  
  
   
 path = []  
 current\_cell = goal\_cell  
 while current\_cell in previous\_dict:  
 path.append(current\_cell)  
 current\_cell = previous\_dict[current\_cell]  
 path.append(start\_cell)  
 path.reverse()  
 print("dijkstra", "length", len(path), path)  
 return path

Эта функция находит кратчайшее расстояние между двумя точками по алгоритму Дейкстры.

Алгоритм начинается с инициализации словаря distance\_dict, который содержит дистанции от начальной клетки до каждой клетки на карте. Дистанции устанавливаются равными бесконечности для каждой клетки, кроме стартовой, которой устанавливается ноль. Словарь previous\_dict создан для хранения предыдущей клетки в кратчайшем пути до каждой клетки. Создается очередь приоритета queue для хранения клеток, необходимых посетить. Очередь инициализируется с начальной клетки и ее расстоянием. Затем алгоритм переходит в цикл, который выполняется до тех пор, пока очередь не опустеет или не будет найдена целевая ячейка. В каждой итерации цикла, клетка с кратчайшим расстоянием от стартовой клетки удаляется из очереди. Алгоритм проверяет соседние клетки для текущей клетки, пропуская не валидные и препятствия. Для каждого валидного соседа алгоритм считает расстояние до этого соседа от начальной клетки и обновляет расстояние и предыдущую клетки для соседа, если это необходимо. Затем обновленный сосед добавляется в очередь. Когда целевая точка достигнута, алгоритм создает кратчайший путь от стартовой клетки до целевой в обратном порядке, поэтому в конце мы используем reverse().

1. **А\***
2. def astar(map\_array, start\_cell, goal\_cell):  
      
    distance\_dict = {}  
    for row in range(MAP\_HEIGHT):  
    for column in range(MAP\_WIDTH):  
    distance\_dict[(column, row)] = float('inf')  
    distance\_dict[start\_cell] = 0  
     
      
    previous\_dict = {}  
     
      
    queue = [(0, start\_cell)]  
     
    while queue:  
      
    current\_distance, current\_cell = heapq.heappop(queue)  
     
      
    if current\_cell == goal\_cell:  
    break  
     
      
    for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:  
    neighbor\_cell = (current\_cell[0] + dx, current\_cell[1] + dy)  
     
      
    if neighbor\_cell[0] < 0 or neighbor\_cell[0] >= MAP\_WIDTH or neighbor\_cell[1] < 0 or neighbor\_cell[  
    1] >= MAP\_HEIGHT:  
    continue  
     
      
    if map\_array[neighbor\_cell[1]][neighbor\_cell[0]][2]:  
    continue  
     
      
    neighbor\_distance = current\_distance + distance(current\_cell, neighbor\_cell)  
     
      
    if goal\_cell != None:  
    heuristic\_distance = evkl\_distance(neighbor\_cell, goal\_cell)  
    else:  
    continue  
     
      
    if neighbor\_distance + heuristic\_distance < distance\_dict[neighbor\_cell]:  
    distance\_dict[neighbor\_cell] = neighbor\_distance + heuristic\_distance  
    previous\_dict[neighbor\_cell] = current\_cell  
    heapq.heappush(queue, (distance\_dict[neighbor\_cell], neighbor\_cell))  
     
      
      
    path = []  
    current\_cell = goal\_cell  
    while current\_cell in previous\_dict:  
    path.append(current\_cell)  
    current\_cell = previous\_dict[current\_cell]  
    path.append(start\_cell)  
    path.reverse()  
    print("astar", "lengh", len(path), path)  
    return path

Эта функция реализует алгоритм поиска A\* для поиска кратчайшего пути между двумя ячейками на карте.

Код сначала создает словарь для хранения расстояния до каждой ячейки, устанавливая расстояние всех ячеек на бесконечность, за исключением начальной ячейки, которая устанавливается на 0. Он также создает пустой словарь для хранения предыдущей ячейки в кратчайшем пути до каждой ячейки и приоритетную очередь для хранения ячеек, которые необходимо посетить, инициализированную начальной ячейкой.

Затем код входит в цикл, который повторяет выбор ячейки с наименьшим расстоянием от начальной ячейки из приоритетной очереди и проверяет ее соседей. Если сосед является допустимой ячейкой (т.е. находится в пределах карты) и не является препятствием, код вычисляет расстояние до соседа как сумму расстояния до текущей ячейки и расстояния между текущей ячейкой и соседом. Он также вычисляет эвристическую оценку стоимости от соседа до целевой ячейки, используя евклидово расстояние, если указана целевая ячейка. Если сумма расстояния и эвристической оценки меньше текущего расстояния до соседа, код обновляет расстояние и предыдущую ячейку для соседа и добавляет его в приоритетную очередь.

Как только цикл завершен, код строит кратчайший путь от начальной ячейки до целевой, начиная с целевой ячейки и следуя предыдущей ячейке для каждой ячейки на пути, пока не достигнет начальной ячейки.

**3.RRT**

def RRT(map\_array, start\_cell, goal\_cell):  
 max\_iterations = 5000  
 step\_size = 1  
 goal\_radius = 1  
 nodes = [start\_cell]  
 edges = []  
 current\_node = start\_cell  
 iterations = 0  
 while current\_node != goal\_cell and iterations < max\_iterations:  
 random\_point = (random.randint(0, MAP\_WIDTH - 1), random.randint(0, MAP\_HEIGHT - 1))  
 nearest\_node = nodes[0]  
 for node in nodes:  
 if evkl\_distance(node, random\_point) < evkl\_distance(nearest\_node, random\_point):  
 nearest\_node = node  
 dx = random\_point[0] - nearest\_node[0]  
 dy = random\_point[1] - nearest\_node[1]  
  
 if abs(dx) > abs(dy):  
 dx = step\_size \* math.copysign(1, dx)  
 dy = 0  
 else:  
 dx = 0  
 dy = step\_size \* math.copysign(1, dy)  
 new\_node = (int(nearest\_node[0] + dx), int(nearest\_node[1] + dy))  
 if new\_node[0] < 0 or new\_node[0] >= MAP\_WIDTH or new\_node[1] < 0 or new\_node[1] >= MAP\_HEIGHT:  
 continue  
 if is\_collision\_free(nearest\_node, new\_node):  
 nodes.append(new\_node)  
 edges.append((nearest\_node, new\_node))  
 current\_node = new\_node  
 if evkl\_distance(current\_node, goal\_cell) < goal\_radius:  
 nodes.append(goal\_cell)  
 edges.append((current\_node, goal\_cell))  
 path = [goal\_cell]  
 while path[-1] != start\_cell:  
 for edge in edges:  
 if edge[1] == path[-1]:  
 path.append(edge[0])  
 break  
 path.reverse()  
 print("edges",edges)  
 print("Path", path)  
 return path  
 iterations += 1  
 return None

Алгоритм строит дерево связанных узлов, начиная с начальной ячейки, и пытается расти деревом к ячейке цели, выбирая случайные точки и соединяя их с ближайшим узлом на дереве, который не пересекается с препятствиями.

max\_iterations: максимальное количество итераций, которое алгоритм выполнит, прежде чем сдаться, если он не может достичь ячейки цели.

step\_size: расстояние между смежными узлами на дереве.

goal\_radius: расстояние, в пределах которого должна быть достигнута ячейка цели, для завершения алгоритма.

nodes: список всех узлов на дереве, изначально содержащий только начальную ячейку.

edges: список всех ребер, соединяющих узлы на дереве, изначально пустой.

current\_node: последний добавленный узел на дереве, изначально начальная ячейка.

iterations: текущее количество итераций, изначально 0.

Основной цикл алгоритма выполняется до тех пор, пока не будет выполнено одно из двух условий: либо текущий узел достигнет ячейки цели, либо будет достигнуто максимальное количество итераций. На каждой итерации выбирается случайная точка из массива карты, находится ближайший узел на дереве с помощью метрики Евклида. Затем создается новый узел, переместившись на расстояние step\_size от ближайшего узла в направлении случайной точки, и этот новый узел добавляется в дерево, если он не пересекается с препятствиями. Если новый узел находится в пределах goal\_radius от ячейки цели, то ячейка цели добавляется в дерево, и путь строится, следуя по ребрам от ячейки цели к начальной ячейке. Если путь не найден после достижения максимального количества итераций, функция возвращает None.