|  |
| --- |
|  |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** |

Институт Искусственного Интеллекта

Кафедра проблем управления

**ЛАБОРАТОРНАЯ РАБОТА №2**

**Вариант №4**

Управление интеллектуальными роботами и робототехническими системами

**Выполнили: Корчагин А.С.**

**Никашин А.К.**

**Группа: КРБО-03-19**

**Преподаватель: Голубов В.В.**

Москва 2023

**Цель работы**: дискредитировать карту, и применить на ней алгоритмы поиска пути

**Описание данных**

Набор исходных данных представляет собой текстовый документ, в котором представлена выборка данных с телеметрии робота Kuka. В последствии преобразованных в карту пространства, на котором были обозначены препятствия.

**Ход работы**

Поскольку данная работа включала в себя два основных раздела, а именно, раздел с дискретизацией пространства и раздел с отработкой алгоритмов. Основной задачей стало дискредитировать пространство для наиболее оптимального перемещения омниколёсного робота по нему.

Так как пространство строилось для омникалёстного робота способного перемещаться во всех направлениях, в пределах двумерной плоскости. Данное свойство робота значительно облегчает принцип построения конфигурационного пространства.

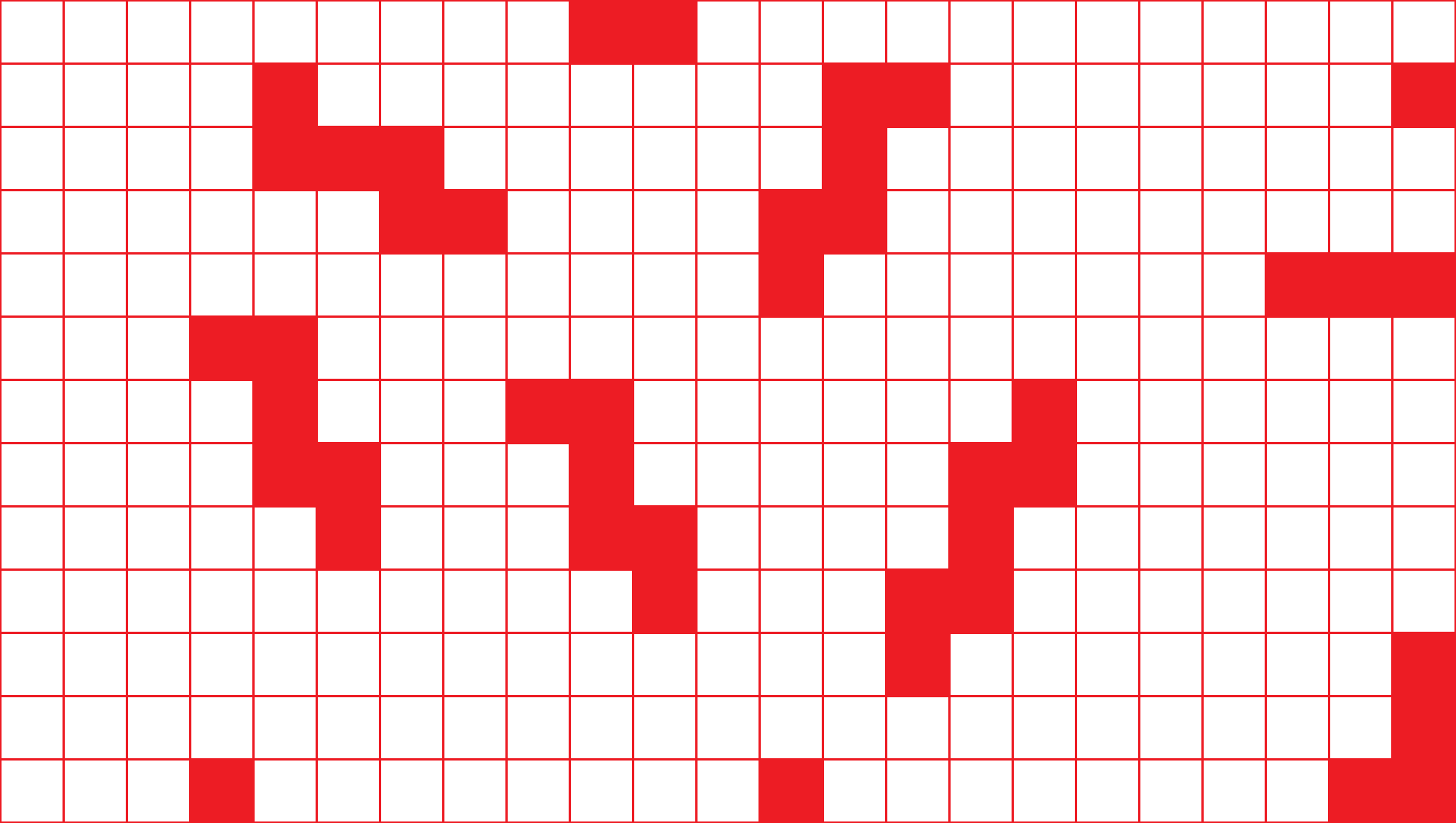


Рис.1 Изображение дискредитированного пространства

Основной прицеп при построении пространства стали, создание карты в которой препятствия уже имеют чёткий контур

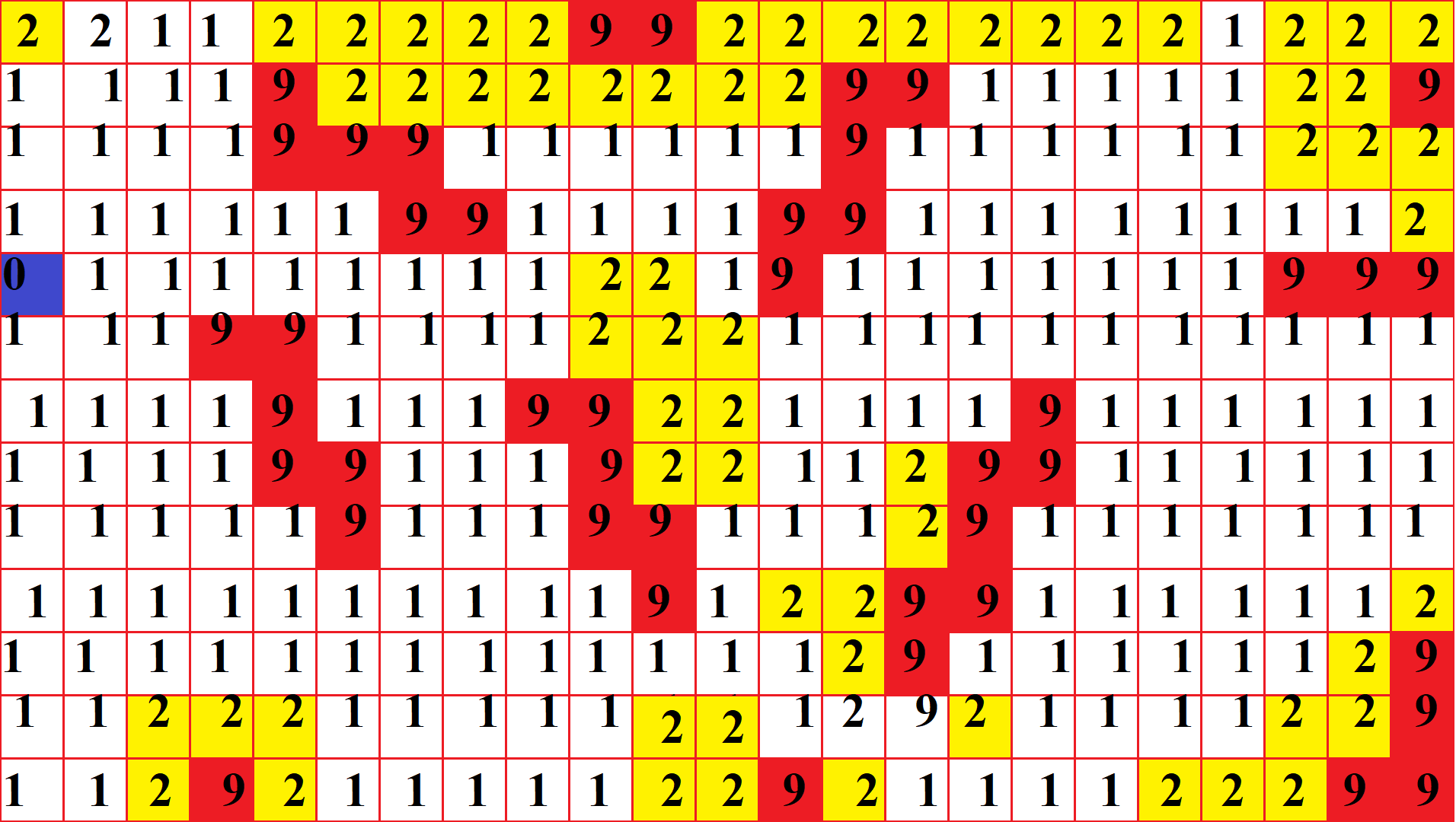
. 

Рис.2 Изображение конфигурационного пространства

Цены перемещений к вершинам графа, заданы списком:

['22112222299222222221222',  
 '11119222222229911111229',  
 '11119991111119111111222',  
 '11111199111199111111112',  
 '01111111122191111111999',  
 '11199111122211111111111',  
 '11119111992211119111111',  
 '11119911192211299111111',  
 '11111911199111291111111',  
 '11111111119122991111112',  
 '11111111111112911111129',  
 '11222111111112921111229',  
 '11292111112292111122299']

При дискретизации пространства, учитывались размеры робота, что позволило принять его за материальную точку. Благодаря этому алгоритмы поиска пути, изначально были упрощены.

Переходя к второй части, необходимо отметить, что после дескритезации, стоит начинать применять алгоритмы. Основных алгоритмов, использованных в работе всего три, первый алгоритм Дейкстры, второй это А\* и последний из алгоритмов это RRT.

Для начала был запущен алгоритм Дейкстра, который позволил построить маршрут из начальной и конечной точек координат, заданных случайно.

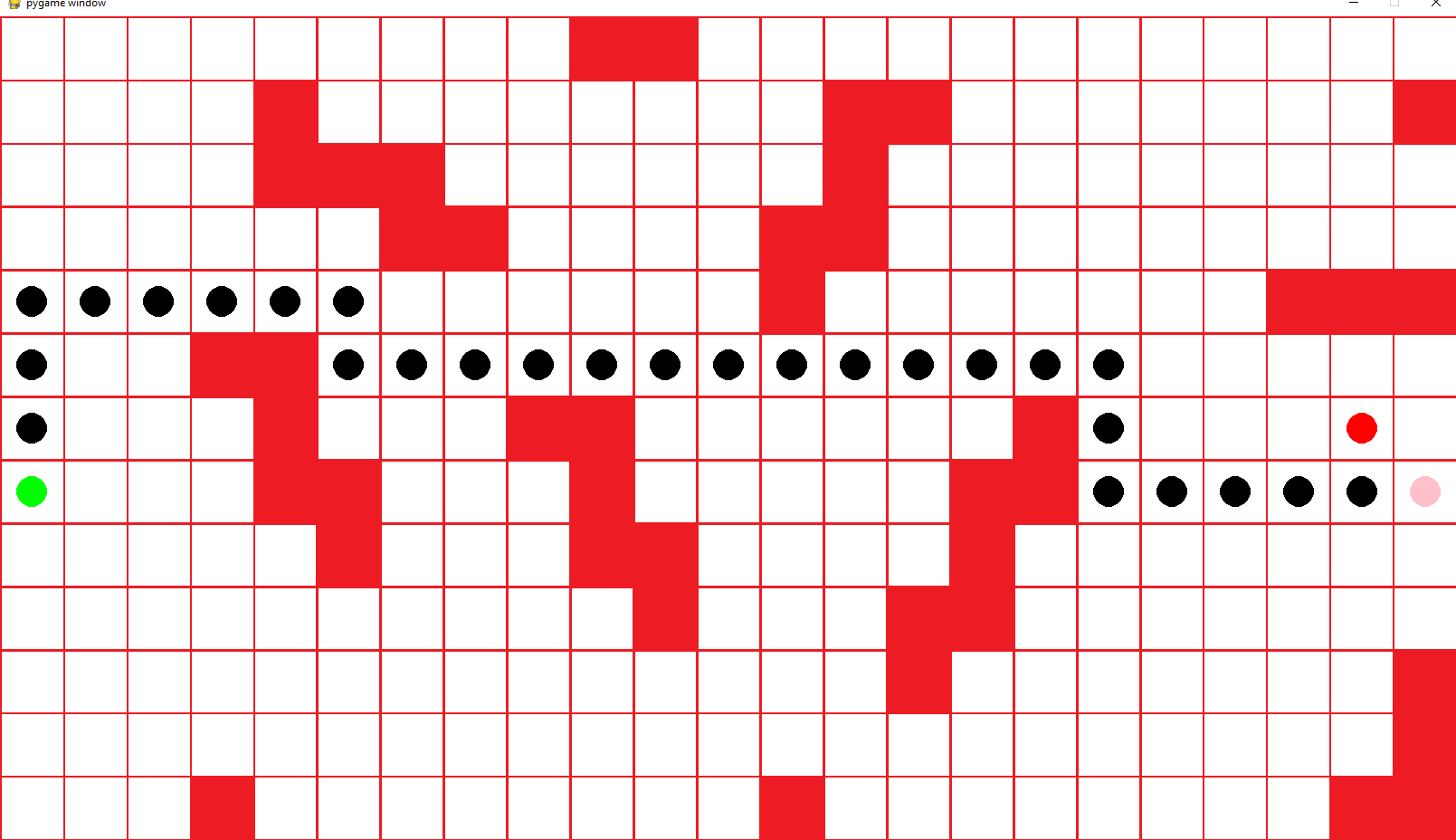


Рис.3 Изображение работы алгоритма Дейстра тест 1

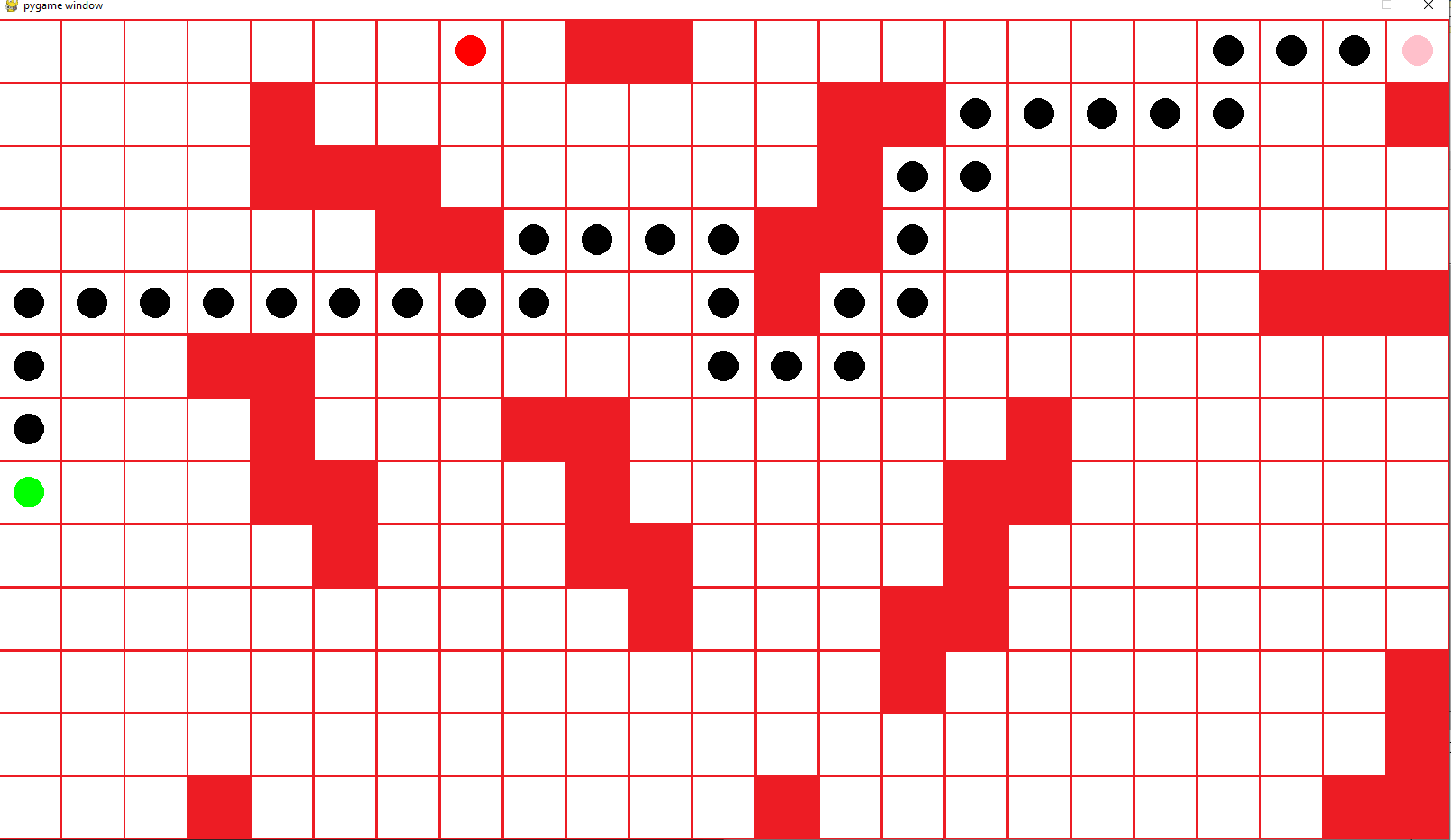


Рис.4 Изображение работы алгоритма Дейстра тест 2

Алгоритм Дейкстры применяется для взвешенного графа в том случае, когда надо найти пути до всех вершин в графе

Итоге данный алгоритм отработал штатно и показал успешный маршрут для случайной точки конца пути.

Затем испытания работы системы были проведены с алгоритмом А\*

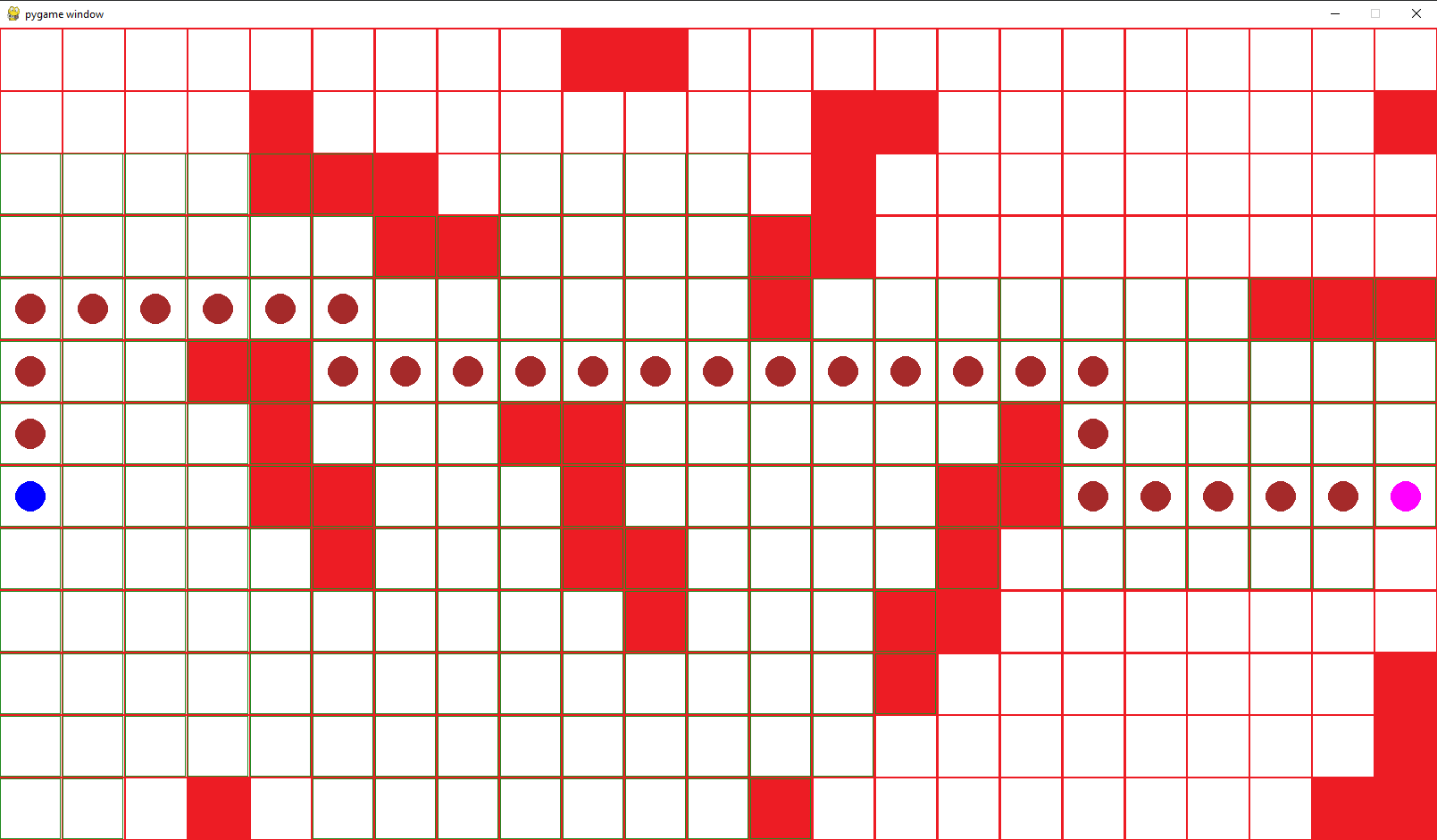
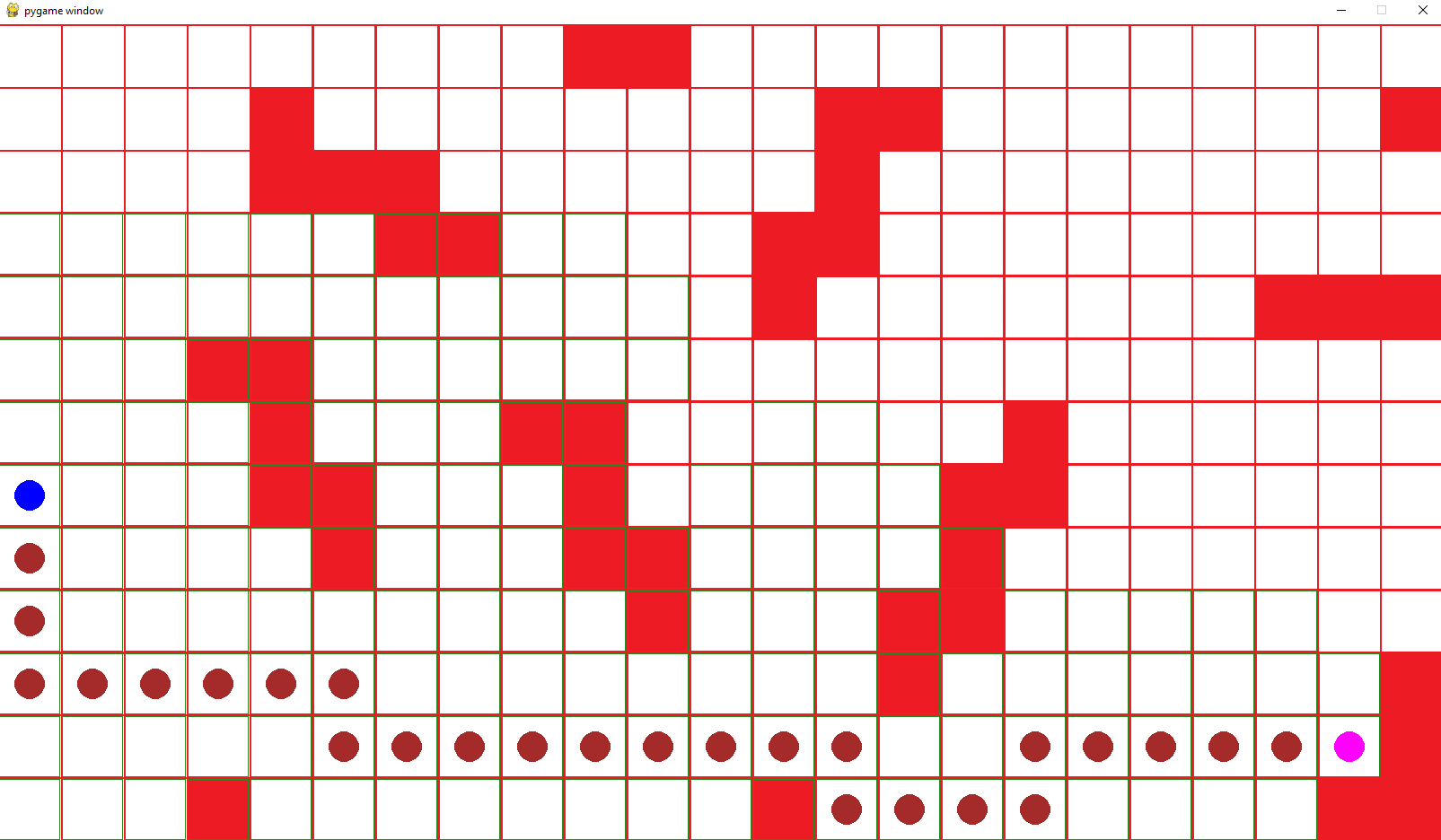


Рис.5 Изображение работы алгоритма A\* тест 1

Рис.6 Изображение работы алгоритма A\* тест 2

И алгоритм А\* идеально отработал в поиске пути для определенной вершины графа.

**Вывод**

В данной лабораторной работе были изучены прицепы построения пространства для работы алгоритмов поиска пути. А так же применены алгоритмы, и установлены наиболее быстрые из них.

Листинг Алгоритм Дейкстры

import pygame as pg  
from heapq import \*  
  
  
def get\_circle(x, y):  
 return (x \* TILE + TILE // 2, y \* TILE + TILE // 2), TILE // 4  
  
  
def get\_neighbours(x, y):  
 check\_neighbour = lambda x, y: True if 0 <= x < cols and 0 <= y < rows else False  
 ways = [-1, 0], [0, -1], [1, 0], [0, 1]#, [-1, -1], [1, -1], [1, 1], [-1, 1]  
 return [(grid[y + dy][x + dx], (x + dx, y + dy)) for dx, dy in ways if check\_neighbour(x + dx, y + dy)]  
  
  
def get\_click\_mouse\_pos():  
 x, y = pg.mouse.get\_pos()  
 grid\_x, grid\_y = x // TILE, y // TILE  
 pg.draw.circle(sc, pg.Color('red'), \*get\_circle(grid\_x, grid\_y))  
 click = pg.mouse.get\_pressed()  
 return (grid\_x, grid\_y) if click[0] else False  
  
  
def heuristic(a, b):  
 return abs(a[0] - b[0]) + abs(a[1] - b[1])  
  
def dijkstra(start, goal, graph):  
 queue = []  
 heappush(queue, (0, start))  
 cost\_visited = {start: 0}  
 visited = {start: None}  
  
 while queue:  
 cur\_cost, cur\_node = heappop(queue)  
 if cur\_node == goal:  
 break  
  
 neighbours = graph[cur\_node]  
 for neighbour in neighbours:  
 neigh\_cost, neigh\_node = neighbour  
 new\_cost = cost\_visited[cur\_node] + neigh\_cost  
  
 if neigh\_node not in cost\_visited or new\_cost < cost\_visited[neigh\_node]:  
 priority = new\_cost + heuristic(neigh\_node, goal)  
 heappush(queue, (priority, neigh\_node))  
 cost\_visited[neigh\_node] = new\_cost  
 visited[neigh\_node] = cur\_node  
 return visited  
  
  
cols, rows = 23, 13  
TILE = 70  
  
pg.init()  
sc = pg.display.set\_mode([cols \* TILE, rows \* TILE])  
clock = pg.time.Clock()  
  
grid = ['22112222299222222221222',  
 '11119222222229911111229',  
 '11119991111119111111222',  
 '11111199111199111111112',  
 '01111111122191111111999',  
 '11199111122211111111111',  
 '11119111992211119111111',  
 '11119911192211299111111',  
 '11111911199111291111111',  
 '11111111119122991111112',  
 '11111111111112911111129',  
 '11222111111112921111229',  
 '11292111112292111122299']  
grid = [[int(char) for char in string ] for string in grid]  
  
graph = {}  
for y, row in enumerate(grid):  
 for x, col in enumerate(row):  
 graph[(x, y)] = graph.get((x, y), []) + get\_neighbours(x, y)  
  
start = (0, 7)  
goal = start  
queue = []  
heappush(queue, (0, start))  
visited = {start: None}  
  
bg = pg.image.load('test1.png').convert()  
bg = pg.transform.scale(bg, (cols \* TILE, rows \* TILE))  
while True:  
  
 sc.blit(bg, (0, 0))  
  
  
 mouse\_pos = get\_click\_mouse\_pos()  
 if mouse\_pos:  
 visited = dijkstra(start, mouse\_pos, graph)  
 goal = mouse\_pos  
  
  
 path\_head, path\_segment = goal, goal  
 while path\_segment and path\_segment in visited:  
 pg.draw.circle(sc, pg.Color('black'), \*get\_circle(\*path\_segment))  
 path\_segment = visited[path\_segment]  
 pg.draw.circle(sc, pg.Color('green'), \*get\_circle(\*start))  
 pg.draw.circle(sc, pg.Color('pink'), \*get\_circle(\*path\_head))  
  
  
 [exit() for event in pg.event.get() if event.type == pg.QUIT]  
 pg.display.flip()  
 clock.tick(30)

Алгоритм А\*

import pygame as pg  
from heapq import \*  
##################################################  
  
def get\_circle(x, y):  
 return (x \* TILE + TILE // 2, y \* TILE + TILE // 2), TILE // 4  
###################################################  
  
def get\_rect(x, y):  
 return x \* TILE + 1, y \* TILE + 1, TILE - 2, TILE - 2  
###################################################  
  
def get\_next\_nodes(x, y):  
 check\_next\_node = lambda x, y: True if 0 <= x < cols and 0 <= y < rows else False  
 ways = [-1, 0], [0, -1], [1, 0], [0, 1]  
 return [(grid[y + dy][x + dx], (x + dx, y + dy)) for dx, dy in ways if check\_next\_node(x + dx, y + dy)]  
  
###################################################  
def heuristic(a, b):  
 return abs(a[0] - b[0]) + abs(a[1] - b[1])  
  
#####################параметры карты###########################  
cols, rows = 23, 13  
TILE = 70  
###################################################  
pg.init()  
sc = pg.display.set\_mode([cols \* TILE, rows \* TILE])  
clock = pg.time.Clock()  
#################Цены перемещений и обозначения препятствий на сетке#############################  
grid = ['22112222299222222221222',  
 '11119222222229911111229',  
 '11119991111119111111222',  
 '11111199111199111111112',  
 '01111111122191111111999',  
 '11199111122211111111111',  
 '11119111992211119111111',  
 '11119911192211299111111',  
 '11111911199111291111111',  
 '11111111119122991111112',  
 '11111111111112911111129',  
 '11222111111112921111229',  
 '11292111112292111122299']  
###################################################  
grid = [[int(char) for char in string ] for string in grid]  
# dict of adjacency lists  
graph = {}  
for y, row in enumerate(grid):  
 for x, col in enumerate(row):  
 graph[(x, y)] = graph.get((x, y), []) + get\_next\_nodes(x, y)  
 ###################################################  
  
################Начальные условия старты и конца пути##########################  
start = (0, 7)  
goal = (21, 11)  
queue = []  
heappush(queue, (0, start))  
cost\_visited = {start: 0}  
visited = {start: None}  
###################подкрузка фона(сетка)########################  
bg = pg.image.load('test1.png').convert()  
bg = pg.transform.scale(bg, (cols \* TILE, rows \* TILE))  
#######################заполнение карты############################  
while True:  
 sc.blit(bg, (0, 0))  
 [pg.draw.rect(sc, pg.Color('forestgreen'), get\_rect(x, y), 1) for x, y in visited]  
 [pg.draw.rect(sc, pg.Color('darkslategray'), get\_rect(\*xy)) for \_, xy in queue]  
 pg.draw.circle(sc, pg.Color('purple'), \*get\_circle(\*goal))  
 ################модификация логики алгоритма Dijkstra до A\*##############################  
  
 if queue:  
 cur\_cost, cur\_node = heappop(queue)  
 if cur\_node == goal:  
 queue = []  
 continue  
  
 next\_nodes = graph[cur\_node]  
 for next\_node in next\_nodes:  
 neigh\_cost, neigh\_node = next\_node  
 new\_cost = cost\_visited[cur\_node] + neigh\_cost  
  
 if neigh\_node not in cost\_visited or new\_cost < cost\_visited[neigh\_node]:  
 priority = new\_cost + heuristic(neigh\_node, goal)  
 heappush(queue, (priority, neigh\_node))  
 cost\_visited[neigh\_node] = new\_cost  
 visited[neigh\_node] = cur\_node  
  
  
 path\_head, path\_segment = cur\_node, cur\_node  
 while path\_segment:  
 pg.draw.circle(sc, pg.Color('brown'), \*get\_circle(\*path\_segment))  
 path\_segment = visited[path\_segment]  
 pg.draw.circle(sc, pg.Color('blue'), \*get\_circle(\*start))  
 pg.draw.circle(sc, pg.Color('magenta'), \*get\_circle(\*path\_head))  
 # pygame necessary lines  
 [exit() for event in pg.event.get() if event.type == pg.QUIT]  
 pg.display.flip()  
 clock.tick(7)