CODE:

```
from functools import total_ordering
from typing import Generator
import pygame
import time

G = 100
O = 0
V = 1
ROB = 3
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
YELLOW = (255, 255, 0)
GREEN = (0, 255, 0)
RED = (255, 0, 0)
WIDTH = 50
HEIGHT = 50
MARGIN = 5
WIND_H = 500
WIND_W = 500

def animate_grid(grid, m, n, path, title, freq=5, obs=O, void=V, rob=ROB):
      pygame.init()
      WINDOW_SIZE = [(WIDTH + MARGIN + 1) * n, (HEIGHT + MARGIN + 1) * m]
      screen = pygame.display.set_mode(WINDOW_SIZE)
      pygame.display.set_caption(title)
      done = False
      clock = pygame.time.Clock()
      path_iter = iter(path)
      while not done:
      try:
            i, j = path_iter.__next__()
      except StopIteration as e:
            return
      grid.graph[i * n + j] = rob
      for event in pygame.event.get():
            done = (event.type == pygame.QUIT)
      screen.fill(BLACK)
      for row in range(m):
            for column in range(n):
                  color = WHITE
                  if grid[row][column] == rob:
                  color = GREEN
                  elif grid[row][column] == obs:
                  color = RED
                  elif grid[row][column] == G:
                  color = YELLOW
                  pygame.draw.rect(screen,
                              color,
                              [(MARGIN + WIDTH) * column + MARGIN,
                                    (MARGIN + HEIGHT) * row + MARGIN,
```

```
                                WIDTH,
                                HEIGHT])
      clock.tick(freq)
      grid.graph[i * n + j] = void
      pygame.display.flip()
      pygame.quit()

@total_ordering
class Point(complex):
      def __init__(self, x, y):
            self.x = x
            self.y = y
            self.tup = (self.x, self.y)
      def __lt__(self, oth):
            return abs(self) < abs(oth)
      def __eq__(self, oth):
            return ((self.x ^ oth.x) + (self.y ^ oth.y)) == 0
      def __iter__(self):
            return self.tup.__iter__() # alternatively iter(self.tup)
      def __str__(self):
            return self.tup.__str__()
      def __hash__(self):
            return self.tup.__hash__()
      def __sub__(self, oth):
            return Point(self.x - oth.x, self.y - oth.y)
      def __str__(self):
            return self.tup.__str__()
      __repr__ = __str__


class Graph:
      def __init__(self, graph, m, n):
            self.graph = graph
            self.m = m
            self.n = n

      def __getitem__(self, indices):
      if type(indices) == type(0):
            i = indices
            return self.graph[i * self.n : (i + 1) * (self.n)]
      elif type(indices) == type(Point(0, 0)):
            i, j = indices
      else:
            if len(indices) == 2:
                  i, j = indices
            else:
                  raise NotImplementedError("Atmost 2 dimensions indexing possible.")
      if j >= n:
            raise IndexError('graph index out of bounds')
      if i < 0 or j < 0:
            raise IndexError("No negative indexing allowed.")
      return self.graph[i * self.n + j]
```

```
def in_unit(p1, p2):
      x, y = map(abs, p1 - p2)
      if x == 1 and y == 1:
            return True
      if (x + y) == 1:
            return True
      return False


def get_adjs(graph: Graph, point: Point, conds=bool, diagonals=False):
      x, y = point
      points = [(x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)]
      if diagonals:
            points.extend(((x + 1, y + 1), (x + 1, y - 1), (x - 1, y - 1), (x - 1, y +
1)))
      avail = map(lambda x: Point(*x), points)
      for pt in avail:
            try:
                  graph[pt]
                  if conds(pt):
                        yield pt
      except Exception as e:
            # print(e)
            pass


def circum_navigate(graph:Graph, init_pos, init_obs_pos, O):
      # O is representation of obstacle in the graph
      visited = {}
      curr_pos = init_pos
      start = True
      path = []
      while (curr_pos != init_pos) or start:
            start = False
            path.append(curr_pos)
            adjs = tuple(get_adjs(graph, curr_pos, lambda x: (not visited.get(x, False))
and in_unit(x, curr_pos) and graph[x] != O))
            for cell in adjs:
                  ans = None
                  if any(graph[ad] == O for ad in get_adjs(graph, cell, diagonals=True)):
                              ans = cell
                              visited[ans] = True
                              break
                  if ans is not None:
                        curr_pos = ans
                  else:
                        break
      return path[1:] + [curr_pos]
```

```
def _BUG(graph, q_start, q_goal, which_bug, obstacle=O, goal=G, void=V):
      visited = {}
      conditions = lambda x: (not visited.get(x, False))
      current_point = q_start
      traced_path = [q_start]
      while graph[current_point] != G:
      adjacents = tuple(get_adjs(graph, current_point, conditions))
      next_point = min(adjacents, key=lambda x: (x - q_goal))
      if graph[next_point] == O:
            circum_points = circum_navigate(graph, current_point, next_point, O)
            min_point = min(circum_points, key=lambda x: abs(q_goal - x))
            if which_bug == 1:
                  traced_path.extend(circum_points)
            elif which_bug == 2:
                        pass
            else:
                        raise NotImplementedError("only bug1 and bug2 supported")
            traced_path.extend(circum_points[: circum_points.index(min_point) + 1])
            next_point = traced_path[-1]
      else:
            traced_path.append(next_point)
      current_point = next_point
      return traced_path


def BUG1(graph, q_start, q_goal, obstacle=O, goal=G, void=V):
      return _BUG(graph, q_start, q_goal, 1, obstacle, goal, void)


def BUG2(graph, q_start, q_goal, obstacle=O, goal=G, void=V):
      return _BUG(graph, q_start, q_goal, 2, obstacle, goal, void)


if __name__ == '__main__':
      graph = [
            V, V, V, V, V, V, V,
            V, V, V, V, V, V, V,
            V, V, O, O, V, V, V,
            V, V, V, O, O, V, V,
            V, V, V, O, O, V, V,
            V, V, V, O, O, V, V,
            V, V, V, V, V, V, V,
            V, V, V, V, V, V, G
      ]

      n = 7
      m = len(graph) // n
      if n * m != len(graph):
            raise ValueError("dimensions doesn't match, graph must be a rectangular
matrix.")

      g = Graph(graph, m, n)
```

```
i = graph.index(G)
path_bug1 = BUG1(g, Point(0, 0), Point(*divmod(i, n)))
path_bug2 = BUG2(g, Point(0, 0), Point(*divmod(i, n)))
print("BUG1 path:", path_bug1)
animate_grid(g, m, n, path_bug1, "BUG1 algorithm")
time.sleep(3)
animate_grid(g, m, n, path_bug2, "BUG2 algorithm")
```

OUTPUT:

```
BUG1 path: [(0, 0), (1, 0), (2, 0), (2, 1), (3, 1), (3, 2), (4, 2), (5, 2), (6, 2), (6, 3), (6, 4), (6, 5), (5, 5), (4, 5), (3, 5), (2, 5), (2, 4), (1, 4), (1, 3), (1, 2), (1, 1), (2, 1), (3, 1), (3, 2), (4, 2), (5, 2), (6, 2), (6, 3), (6, 4), (6, 5), (7, 5), (7, 6)]
```