HW3

March 7, 2022

This uses the same A* algorithm as HW1 on the GVD path Brushfire creates. The A* is limited to only Single-sided searching rather than Double like before, since there's no practical advantage otherwise in the case of exploring a GVD. There are only a few paths to explore, and a path from start to goal is already guaranteed to exist. It should also not use Jump Point since that paths are narrow enough that jumping from corners won't make a significant change to path length while still adding more time for processing. But I left it in because I think it's interesting to see.

In regards to the Jump Point algorithm I did need to make a slight tweak though. Because the paths are so narrow it's possible that two corner nodes may not be visible to each other even though a path between them exists. In this case, the algorithm would be unable to continue and would just get stuck in a loop. To prevent this I updated the definition of a corner node to include not just nodes diagonally adjacent to outside corners, but nodes diagonally adjacent to inside corners as well. For example, if an obstacle were L-shaped, the corner nodes ($*$) would be both the node outside and the node inside.

```
 |
 |
 |_*_  __  __
*
```

This should prevent any errors in the algorithm, though it can also increase search time as the number of nodes to explore may also increase.

1 Brushfire

brushfire.py

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

show_animation = True


class Brushfire:
    def __init__(self, grid):
        self.grid = grid
        global show_animation
        show_animation = grid.show
        self.minX = grid.r1[0]
```

```
        self.minY = grid.r2[0]
        self.gvd = [[self.Node(0, x, y, [x, y]) for y in grid.r2] for x in grid.
↪r1]
        self.moves = [0, 1,
                      1, 0,
                      0, -1,
                      -1, 0,
                      1, 1,
                      1, -1,
                      -1, -1,
                      -1, 1]
        self.get_gvd()

    class Node:
        def __init__(self, val, x, y, parent_index):
            self.val = val
            self.x = x
            self.y = y
            self.parent_index = parent_index

        def __eq__(self, n2):
            return self.val == n2.val

        def same_parent(self, node):
            return self.parent_index == node.parent_index

    def get_gvd(self):
        open_set = []
        temp_set = []

        for i in range(len(self.grid.r1)):
            x = i + self.minX
            for j in range(len(self.grid.r2)):
                y = j + self.minY
                node = self.gvd[i][j]
                if not self.grid.obstacle_map[x][y]:
                    open_set.append(node)
                else:
                    node.val = 1
                    temp_set.append(node)
                    for k in range(len(self.moves) // 4):
                        x2 = i + self.moves[2 * k]
                        y2 = j + self.moves[2 * k + 1]
                        if len(self.grid.r1) > x2 > -1 and len(self.grid.r2) >␣
↪y2 > -1:
                            if self.grid.obstacle_map[x2 + self.minX][y2 + self.
↪minY]:
```

```
                                self.gvd[x2][y2].parent_index = node.
↪parent_index
        if show_animation:
            line = plt.plot(0, 0)
        while len(open_set) > 0:
            target_set = temp_set.copy()
            temp_set = []
            for node in target_set:
                for i in range(len(self.moves) // 2):
                    x = node.x + self.moves[2 * i]
                    y = node.y + self.moves[2 * i + 1]
                    if len(self.grid.r1) > x - self.minX > -1 and len(self.grid.
↪r2) > y - self.minY > -1:
                        neighbor = self.gvd[x - self.minX][y - self.minY]
                        if neighbor.val == 0:
                            self.grid.obstacle_map[x][y] = True
                            neighbor.parent_index = node.parent_index
                            neighbor.val = node.val + 1
                            temp_set.append(neighbor)
                            open_set.remove(neighbor)
                            # if show_animation:
                                # line = plt.plot(x, y, "s", color=line[0].
↪get_color())
                        elif neighbor.val >= node.val and not node.
↪same_parent(neighbor):
                            self.grid.obstacle_map[x][y] = False
                            neighbor.val = np.inf
                            if show_animation:
                                plt.plot(x, y, "s", color="#888888")
            if show_animation:
                plt.gcf().canvas.mpl_connect('key_release_event',
                                             lambda event: [exit(0) if event.
↪key == 'escape' else None])
                plt.pause(0.001)
                line = plt.plot(0, 0)

        for node in temp_set:
            for i in range(len(self.moves) // 2):
                x = node.x + self.moves[2 * i]
                y = node.y + self.moves[2 * i + 1]
                if len(self.grid.r1) > x - self.minX > -1 and len(self.grid.r2)␣
↪> y - self.minY > -1:
                    neighbor = self.gvd[x - self.minX][y - self.minY]
                    if neighbor.val >= node.val and not node.
↪same_parent(neighbor):
                        self.grid.obstacle_map[x][y] = False
```

```
                        neighbor.val = np.inf
                        if show_animation:
                            plt.plot(x, y, "s", color="#888888")
        if show_animation:
            plt.pause(0.001)

        self.grid.obstacle_map[self.grid.start[0]][self.grid.start[1]] = False
        self.grid.obstacle_map[self.grid.goal[0]][self.grid.goal[1]] = False

        node = self.gvd[self.grid.start[0] - self.minX][self.grid.start[1] -␣
↪self.minY]
        while node.val != np.inf:
            for i in range(len(self.moves) // 2):
                x = node.x + self.moves[2 * i] - self.minX
                y = node.y + self.moves[2 * i + 1] - self.minY
                neighbor = self.gvd[x][y]
                if neighbor.val > node.val:
                    node = self.gvd[x][y]
                    self.grid.obstacle_map[x + self.minX][y + self.minY] = False
                    if show_animation:
                        plt.plot(x + self.minX, y + self.minX, "s",␣
↪color="#888888")
                    break

        node = self.gvd[self.grid.goal[0] - self.minX][self.grid.goal[1] - self.
↪minY]
        while node.val != np.inf:
            for i in range(len(self.moves) // 2):
                x = node.x + self.moves[2 * i] - self.minX
                y = node.y + self.moves[2 * i + 1] - self.minY
                neighbor = self.gvd[x][y]
                if neighbor.val > node.val:
                    node = self.gvd[x][y]
                    self.grid.obstacle_map[x + self.minX][y + self.minY] = False
                    if show_animation:
                        plt.plot(x + self.minX, y + self.minX, "s",␣
↪color="#888888")
                    break
```

2 A*

aStar.py

[ ]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
```

```python
maze = False
show_animation = True
single_sided_astar = False


class Grid:
    def __init__(self, sx=0, sy=0, gx=0, gy=0, rr=0.0, single=False, m=False, show=True):
        global maze
        global show_animation
        global single_sided_astar
        maze = m
        single_sided_astar = single
        show_animation = show
        self.single = single
        self.maze = m
        self.show = show
        self.start = (sx, sy)
        self.goal = (gx, gy)
        self.rr = rr
        self.width, self.height = 0, 0
        self.r1, self.r2 = [], []
        self.init()
        self.obstacle_map = [[False for _ in self.r2]
                             for _ in self.r1]
        self.motion = [[-1, 0, 1],
                       [0, 1, 1],
                       [1, 0, 1],
                       [0, -1, 1],
                       [-1, -1, math.sqrt(2)],
                       [-1, 1, math.sqrt(2)],
                       [1, 1, math.sqrt(2)],
                       [1, -1, math.sqrt(2)]]

    def copy(self, grid):
        global maze
        global show_animation
        global single_sided_astar
        maze = grid.maze
        single_sided_astar = grid.single
        show_animation = grid.show
        self.single = grid.single
        self.maze = grid.maze
        self.show = grid.show
        self.start = grid.start
        self.goal = grid.goal
```

```
        self.rr = grid.rr
        self.width, self.height = grid.width, grid.height
        self.r1, self.r2 = grid.r1, grid.r2
        self.obstacle_map = grid.obstacle_map
        self.motion = grid.motion

    class Node:
        def __init__(self, x, y, cost, parent_index):
            self.x = x
            self.y = y
            self.cost = cost
            self.parent_index = parent_index

        def __eq__(self, n2):
            return self.x == n2.x and self.y == n2.y

    def aStar(self, start=None, goal=None, checked_set=None, corner_set2=None,␣
↪single=False):
        print("Exploring...")

        if start is None:
            start = self.start
        if goal is None:
            goal = self.goal
        if checked_set is None:
            checked_set = dict()
        if corner_set2 is None:
            corner_set2 = dict()

        start_node = self.Node(self.calc_xy_index(start[0], self.start[0]),
                               self.calc_xy_index(start[1], self.start[1]), 0.
↪0, -1)
        goal_node = self.Node(self.calc_xy_index(goal[0], self.start[0]),
                              self.calc_xy_index(goal[1], self.start[1]), 0.0,␣
↪-1)

        corner_set = dict()
        open_set, closed_set = dict(), dict()
        open_set[self.calc_grid_index(start_node)] = start_node

        open_set2, closed_set2 = dict(), dict()
        open_set2[self.calc_grid_index(goal_node)] = goal_node
        current2, c_id2 = None, None

        while 1:
            if len(open_set)*len(open_set2) == 0:
                print("No path exists")
```

```
                break

            c_id = min(open_set, key=lambda o: open_set[o].cost + self.
↪calc_heuristic(goal_node, open_set[o]))
            current = open_set[c_id]

            if not single:
                c_id2 = min(open_set2, key=lambda o: open_set2[o].cost + self.
↪calc_heuristic(start_node, open_set2[o]))
                current2 = open_set2[c_id2]

            if show_animation:
                if single:
                    plt.plot(self.calc_grid_position(current.x, self.start[0]),
                             self.calc_grid_position(current.y, self.start[1]),␣
↪"1r")
                else:
                    plt.plot(self.calc_grid_position(current.x, self.start[0]),
                             self.calc_grid_position(current.y, self.start[1]),␣
↪"+y")
                    plt.plot(self.calc_grid_position(current2.x, self.start[0]),
                             self.calc_grid_position(current2.y, self.
↪start[1]), "xc")
                # for stopping simulation with the esc key.
                plt.gcf().canvas.mpl_connect('key_release_event',
                                             lambda event: [exit(0) if event.
↪key == 'escape' else None])
                if len(closed_set.keys()) % 10 == 0:
                    plt.pause(0.001)

            exist = False
            if single:
                if current == goal_node:
                    exist = True
            else:
                c_gd = self.calc_grid_index(current)
                c_gd2 = self.calc_grid_index(current2)
                if c_gd in closed_set2:
                    exist = True
                    corner_set[c_gd] = current
                elif c_gd2 in closed_set:
                    exist = True
                    current = current2
                    corner_set[c_gd2] = current2

            if exist and len(corner_set2) == 0:
```

```
                print("A path exists.", len(closed_set) + len(closed_set2) - 1,␣
↪"nodes explored.")
                if show_animation:
                    plt.plot(self.calc_grid_position(current.x, self.start[0]),
                             self.calc_grid_position(current.y, self.start[1]),␣
↪"or")
                corner_set[self.calc_grid_index(start_node)] = start_node
                corner_set[self.calc_grid_index(goal_node)] = goal_node
                if single:
                    return self.jump_point(start_node, goal_node, goal_node,␣
↪corner_set)
                else:
                    return self.jump_point(start_node, current, goal_node,␣
↪corner_set)

            del open_set[c_id]
            closed_set[c_id] = current
            self.expand_grid(current, c_id, open_set, closed_set, corner_set)

            if current != start_node and self.visible(start_node, current):
                for _, node in checked_set.items():
                    if self.visible(node, current):
                        print(len(closed_set) + len(closed_set2) - 1, "nodes␣
↪explored.")
                        corner_set2[c_id] = self.Node(current.x, current.y, np.
↪inf, -1)
                        plt.plot(self.calc_grid_position(current.x, self.
↪start[0]),
                                 self.calc_grid_position(current.y, self.
↪start[1]), "^b")
                        return
                    if c_id in corner_set and c_id not in corner_set2:
                        corner_set2[c_id] = self.Node(current.x, current.y, np.
↪inf, -1)
                        plt.plot(self.calc_grid_position(current.x, self.
↪start[0]),
                                 self.calc_grid_position(current.y, self.
↪start[1]), "^m")

            if not single:
                del open_set2[c_id2]
                closed_set2[c_id2] = current2
                self.expand_grid(current2, c_id2, open_set2, closed_set2,␣
↪corner_set)

    def expand_grid(self, current, c_id, open_set, closed_set, corner_set):
```

```
        obs = []
        for i, _ in enumerate(self.motion):
            node = self.Node(current.x + self.motion[i][0], current.y + self.
↪motion[i][1],
                             current.cost + self.motion[i][2], c_id)
            n_id = self.calc_grid_index(node)

            if self.obstacle_node(node):
                obs.append(i)
                continue

            if self.bounds_node(node) or n_id in closed_set:
                continue

            if n_id not in open_set:
                if maze:
                    if i < 4:
                        open_set[n_id] = node
                else:
                    open_set[n_id] = node
            elif open_set[n_id].cost > node.cost:
                open_set[n_id] = node

        corner = False
        if 4 in obs and 3 not in obs and 0 not in obs or \
                5 in obs and 0 not in obs and 1 not in obs or \
                6 in obs and 1 not in obs and 2 not in obs or \
                7 in obs and 2 not in obs and 3 not in obs:
            corner_set[c_id] = self.Node(current.x, current.y, np.inf, -1)
            corner = True
        elif 4 in obs and 3 in obs and 0 in obs or \
                5 in obs and 0 in obs and 1 in obs or \
                6 in obs and 1 in obs and 2 in obs or \
                7 in obs and 2 in obs and 3 in obs:
            corner_set[c_id] = self.Node(current.x, current.y, np.inf, -1)
            corner = True

        if corner and show_animation:
            plt.plot(self.calc_grid_position(current.x, self.start[0]),
                     self.calc_grid_position(current.y, self.start[1]), "^m")

    def jump_point(self, start, intersect, goal, corner_set):
        print("Finding path...")

        grid = Grid()
        grid.copy(self)
```

```
        marker = "^g"
        found_intersect = not maze

        open_set = dict()
        open_set[self.calc_grid_index(start)] = start
        closed_set, closed_set2 = dict(), corner_set.copy()

        while 1:
            if len(open_set) == 0:
                print("Path not found. Expanding intersect node...")
                grid.aStar((intersect.x, intersect.y), (start.x, start.y),␣
↪closed_set, corner_set, True)
                grid.aStar((intersect.x, intersect.y), (goal.x, goal.y),␣
↪closed_set2, corner_set, True)
                print("Finding path...")
                marker = "^r"
                found_intersect = False
                open_set[self.calc_grid_index(start)] = start
                closed_set, closed_set2 = dict(), corner_set.copy()

            if not found_intersect:
                c_id = min(open_set, key=lambda o: open_set[o].cost + self.
↪calc_heuristic(intersect, open_set[o]))
                current = open_set[c_id]
            else:
                c_id = min(open_set, key=lambda o: open_set[o].cost + self.
↪calc_heuristic(goal, open_set[o]))
                current = open_set[c_id]

            if current == intersect:
                found_intersect = True
                open_set = dict()
                open_set[self.calc_grid_index(intersect)] = intersect

            if show_animation:
                plt.plot(self.calc_grid_position(current.x, self.start[0]),
                         self.calc_grid_position(current.y, self.start[1]),␣
↪marker)
                plt.gcf().canvas.mpl_connect('key_release_event',
                                             lambda event: [exit(0) if event.
↪key == 'escape' else None])
                if len(closed_set.keys()) % 1 == 0:
                    plt.pause(0.001)

            if current == goal:
```

```
                print("Goal Found!", len(closed_set) - 1, "corner nodes explored.")
                self.calc_final_path(current, closed_set)
                return

            del open_set[c_id]
            del closed_set2[c_id]
            closed_set[c_id] = current
            self.expand_corners(current, c_id, open_set, closed_set, corner_set)

    def expand_corners(self, current, c_id, open_set, closed_set, corner_set):
        for _, corner in corner_set.items():
            node = self.Node(corner.x, corner.y, current.cost + self.calc_heuristic(current, corner), c_id)
            n_id = self.calc_grid_index(node)

            if n_id in closed_set or not self.visible(node, current):
                continue
            if n_id not in open_set or open_set[n_id].cost > node.cost:
                open_set[n_id] = node

    def visible(self, node, current):
        rise = node.y - current.y
        run = node.x - current.x

        for x in range(min(current.x, node.x), max(current.x, node.x)):
            m = rise / run
            b = current.y - current.x * m
            y = m * x + b
            if self.obstacle_map[x][math.floor(y)] or self.obstacle_map[x][math.ceil(y)]:
                return False
        for y in range(min(current.y, node.y), max(current.y, node.y)):
            n = run / rise
            d = current.x - current.y * n
            x = n * y + d
            if self.obstacle_map[math.floor(x)][y] or self.obstacle_map[math.ceil(x)][y]:
                return False
        return True

    def calc_final_path(self, goal, closed_set):
        # generate final course
        total = 0
        rx, ry = [self.calc_grid_position(goal.x, self.start[0])], [
            self.calc_grid_position(goal.y, self.start[1])]
        parent_index = goal.parent_index
```

```python
        while parent_index != -1:
            n = closed_set[parent_index]
            rx.append(self.calc_grid_position(n.x, self.start[0]))
            ry.append(self.calc_grid_position(n.y, self.start[1]))
            parent_index = n.parent_index
        for i in range(0, len(rx) - 1):
            total += math.hypot(rx[i] - rx[i + 1], ry[i] - ry[i + 1])
            if show_animation:
                plt.plot(rx[i], ry[i], "ob")
                plt.plot((rx[i], rx[i + 1]), (ry[i], ry[i + 1]), "-b")
                plt.pause(0.001)
        if show_animation:
            plt.plot(self.start[0], self.start[1], "ob")
        print("Path length:", total)

    @staticmethod
    def calc_heuristic(n1, n2):
        return math.hypot(n1.x - n2.x, n1.y - n2.y)

    @staticmethod
    def calc_grid_position(index, min_position):
        return index + min_position

    @staticmethod
    def calc_xy_index(position, min_pos):
        return position - min_pos

    def calc_grid_index(self, node):
        return (node.y - self.start[1]) * self.width + (node.x - self.start[0])

    def obstacle_node(self, node):
        return self.obstacle_map[node.x][node.y]

    def bounds_node(self, node):
        px = self.calc_grid_position(node.x, self.start[0])
        py = self.calc_grid_position(node.y, self.start[1])

        if px < self.start[0] - 5:
            return True
        elif py < self.start[1] - 5:
            return True
        elif px > self.goal[0] + 5:
            return True
        elif py > self.goal[1] + 5:
            return True
        return False
```

```
    def init(self):
        self.width = self.goal[0] - self.start[0] + 10
        self.height = self.goal[1] - self.start[1] + 10
        self.r1 = range(self.start[0] - 5, self.goal[0] + 6)
        self.r2 = range(self.start[1] - 5, self.goal[1] + 6)

    def calc_obstacle_map(self, ox, oy):
        # obstacle map generation
        for ix in self.r1:
            x = self.calc_grid_position(ix, self.start[0])
            for iy in self.r2:
                y = self.calc_grid_position(iy, self.start[1])
                for iox, ioy in zip(ox, oy):
                    d = math.hypot(iox - x, ioy - y)
                    if d <= self.rr:
                        self.obstacle_map[ix][iy] = True
                        break
```

3 Main

main.py

[ ]:
```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.patches import Rectangle
from numpy.random import rand
from aStar import Grid
from brushfire import Brushfire

np.random.seed(7)
show_animation = True
single_sided_astar = True


def main():
    print(__file__ + "  Press Esc to exit")

    # start and goal position
    sx = 0
    sy = 0
    gx = 35
    gy = 35
    robot_radius = 0.5
    grid = Grid(sx, sy, gx, gy, robot_radius, single_sided_astar, False,␣
↪show_animation)
```

```
    if show_animation:
        plt.plot(sx, sy, "og")
        plt.plot(gx, gy, "ob")
        plt.grid(True)
        plt.axis("equal")
        plt.pause(1)

    ox = [x for x in range(sx - 5, gx + 6)]
    oy = [sy - 5 for _ in range(sy - 5, gy + 6)]
    grid.calc_obstacle_map(ox, oy)
    if show_animation:
        plt.plot(ox, oy, "sk")

    ox = [x for x in range(sx - 5, gx + 6)]
    oy = [gy + 5 for _ in range(sy - 5, gy + 6)]
    grid.calc_obstacle_map(ox, oy)
    if show_animation:
        plt.plot(ox, oy, "sk")

    ox = [sx - 5 for _ in range(sx - 5, gx + 6)]
    oy = [y for y in range(sy - 5, gy + 6)]
    grid.calc_obstacle_map(ox, oy)
    if show_animation:
        plt.plot(ox, oy, "sk")

    ox = [gx + 5 for _ in range(sx - 5, gx + 6)]
    oy = [y for y in range(sy - 5, gy + 6)]
    grid.calc_obstacle_map(ox, oy)
    if show_animation:
        plt.plot(ox, oy, "sk")

    # 4 random rectangular obstacles, one in each quadrant
    print("Creating Obstacles...")
    obs = []
    w = gx - sx
    h = gy - sy
    # Quad 1
    ob = Rectangle(xy=(rand() * w / 2 + w / 2 - 7, rand() * h / 2 + h / 2 - 7),␣
↪width=rand() * 8 + 4 + robot_radius,
                   height=rand() * 8 + 4 + robot_radius)
    while ob.contains_point((sx, sy)) or ob.contains_point((gx, gy)):
        ob = Rectangle(xy=(rand() * w / 2 + w / 2 - 7, rand() * h / 2 + h / 2 -␣
↪7), width=rand() * 8 + 4 + robot_radius,
                       height=rand() * 8 + 4 + robot_radius)
    ob.set_width(ob.get_width() - robot_radius)
    ob.set_height(ob.get_height() - robot_radius)
    obs.append(ob)
```

```
    # Quad 2
    ob = Rectangle(xy=(rand() * w / 2, rand() * h / 2 + h / 2 - 7), width=rand() * 8 + 4,
                   height=rand() * 8 + 4)
    obs.append(ob)

    # Quad 3
    ob = Rectangle(xy=(rand() * w / 2, rand() * h / 2), width=rand() * 8 + 4 + robot_radius,
                   height=rand() * 8 + 4 + robot_radius)
    while ob.contains_point((sx, sy)) or ob.contains_point((gx, gy)):
        ob = Rectangle(xy=(rand() * w / 2, rand() * h / 2), width=rand() * 8 + 4 + robot_radius,
                       height=rand() * 8 + 4 + robot_radius)
    ob.set_width(ob.get_width() - robot_radius)
    ob.set_height(ob.get_height() - robot_radius)
    obs.append(ob)

    # Quad 4
    ob = Rectangle(xy=(rand() * w / 2 + w / 2 - 7, rand() * h / 2), width=rand() * 8 + 4,
                   height=rand() * 8 + 4)
    obs.append(ob)

    # discretize each rectangle
    for ob in obs:
        x = ob.get_x()
        y = ob.get_y()

        ox, oy = [], []
        for i in range(int(x), int(x + ob.get_width())):
            for j in range(int(y), int(y + ob.get_height())):
                if ob.contains_point((i, j)):
                    ox.append(i)
                    oy.append(j)
        grid.calc_obstacle_map(ox, oy)
        if show_animation:
            plt.plot(ox, oy, "sk")
            plt.gcf().canvas.mpl_connect('key_release_event',
                                         lambda event: [exit(0) if event.key == 'escape' else None])
            plt.pause(0.01)

    Brushfire(grid)
    grid.aStar(single=single_sided_astar)
    if show_animation:
```

```
        plt.show()


if __name__ == '__main__':
    main()
```