HW1

March 7, 2022

This A* algorithm combines two different types of A*: Double-sided and Jump Point.

Double-sided A* is, as the name implies, an algorithm that uses A* on both the start and goal positions. The advantage of Double-sided over single is that the algorithm will know if a path even exists sooner. In many cases it may even end up exploring less of the map than the normal A* algorithm. However, the Double-sided A* is a bit more complex, and may not offer a significant advantage in an open space map. Though in a maze-like environment, Double-sided A* may be preferable.

Jump Point A* preprocesses the map to find all the corners of the obstacles inside and then performs A* on just those corner positions. This offers the advantage of needing to explore significantly less of the environment to find a path. It can also help to shorten the path length by jumping from position to position regardless of angle, instead of needing to move in strictly Hamming/Euclidean directions one neighbor at a time. In a maze-like environment this may come as a disadvantage though, since there may be about as many corner positions to explore as there are normal positions.

By combining these two, the preprocessing step in Jump Point can be left to Double-sided A*. Doing so eliminates the need to preprocess the entire map and find useless corners, and it also allows Jump Point to know ahead of time if a path even exists. Once done, Jump Point can find a shorter path by exploring just the corner positions that Double-sided A* found rather than all the corners that exist in the map. Additionally, Double A* finds only outside corner nodes. For example, if an obstacle were L-shaped, a corner node (∗) would be the node outside but not the node inside.

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

In this regard, this A* is essentially making a Visibility Graph in a discretized space.

1 Modified A* Algorithm code

aStar.py

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

maze = False
```

```python
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
```

```python
        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

        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)
```

```python
    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
        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] - self.width / 4:
            return True
        elif py < self.start[1] - self.height / 4:
            return True
        elif px > self.goal[0] + self.width / 4:
            return True
        elif py > self.goal[1] + self.height / 4:
            return True
        return False

    def init(self):
        self.width = 2 * (self.goal[0] - self.start[0])
        self.height = 2 * (self.goal[1] - self.start[1])
        self.r1 = range(math.floor(self.start[0] - self.width / 4), math.
↪ceil(self.goal[0] + self.width / 4))
        self.r2 = range(math.floor(self.start[1] - self.height / 4), math.
↪ceil(self.goal[1] + self.height / 4))

    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
```

2 Main

main.py

[ ]:
```
from aStar import Grid
import math
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
from numpy.random import rand

np.random.seed(11)
maze = False
show_animation = True
single_sided_astar = False


def make_maze(top_vertex, bottom_vertex, obs_number=1500):
    """
    Author: Weicent
    https://github.com/AtsushiSakai/PythonRobotics/blob/master/PathPlanning/
↪AStar/a_star_searching_from_two_side.py
    randomly generate maze
    """
    # below can be merged into a rectangle boundary
    ay = list(range(bottom_vertex[1], top_vertex[1]))
    ax = [bottom_vertex[0]] * len(ay)
    cy = ay
    cx = [top_vertex[0]] * len(cy)
    bx = list(range(bottom_vertex[0] + 1, top_vertex[0]))
    by = [bottom_vertex[1]] * len(bx)
    dx = [bottom_vertex[0]] + bx + [top_vertex[0]]
    dy = [top_vertex[1]] * len(dx)

    # generate random obstacles
```

```python
    ob_x = np.random.randint(bottom_vertex[0] + 1, top_vertex[0], obs_number).
↪tolist()
    ob_y = np.random.randint(bottom_vertex[1] + 1, top_vertex[1], obs_number).
↪tolist()
    # x y coordinate in certain order for boundary
    x = ax + bx + cx + dx
    y = ay + by + cy + dy
    obstacle = np.vstack((ob_x, ob_y)).T.tolist()
    obs_array = np.array(obstacle)
    bound = np.vstack((x, y)).T
    bound_obs = np.vstack((bound, obs_array))
    return bound_obs


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, maze,␣
↪show_animation)

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

    if maze:
        print("Creating Obstacles...")
        obs = make_maze((gx + 10, gy + 10), (sx - 10, sy - 10))
        ox, oy = [], []
        for ob in obs:
            if (ob[0] != sx or ob[1] != sy) and (ob[0] != gx or ob[1] != gy):
                ox.append(ob[0])
                oy.append(ob[1])
                if show_animation:
                    if len(ox) % 150 == 0:
                        plt.plot(ox, oy, "sk")
                        plt.gcf().canvas.mpl_connect('key_release_event',
                                                     lambda event: [exit(0) if␣
↪event.key == 'escape' else None])
```

```python
                        plt.pause(0.001)
                        grid.calc_obstacle_map(ox, oy)
                        ox, oy = [], []
        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(0.001)
        grid.calc_obstacle_map(ox, oy)
    else:
        # a bunch of random elliptical obstacles
        obs = []
        w = gx - sx
        h = gy - sy
        for _ in range(15):
            ob = Ellipse(xy=(rand() * w + sx, rand() * h + sy), width=rand() * 16 + 4 + robot_radius,
                         height=rand() * 16 + 4 + robot_radius, angle=rand() * 360)
            while ob.contains_point((sx, sy)) or ob.contains_point((gx, gy)):
                ob = Ellipse(xy=(rand() * w + sx, rand() * h + sy), width=rand() * 16 + 4 + robot_radius,
                             height=rand() * 16 + 4 + robot_radius, angle=rand() * 360)
            ob.width -= robot_radius
            ob.height -= robot_radius
            obs.append(ob)

        # discretize each ellipse
        print("Creating Obstacles...")
        for ob in obs:
            h = math.ceil(max(ob.width, ob.height) / 2)
            x = ob.center[0] - h
            y = ob.center[1] - h

            ox, oy = [], []
            for i in range(int(x), int(x + 2 * h)):
                for j in range(int(y), int(y + 2 * h)):
                    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(0.001)

    grid.aStar(single=single_sided_astar)
    if show_animation:
        plt.show()


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