CS 470: Project 1 - Pathfinding

University of Idaho
Matthew Waltz
9th February 2018

Abstract

A Python implementation illustrates finding various paths from a designated starting position to a final goal, exported visually to png files for ease of verification. The available directions include up, down, left, and right, but exclude diagonal movement. Algorithms implemented include breadth first, lowest cost, greedy best first, and A*. Heuristics used include Manhattan and Euclidean distances. The efficiency and results of these algorithms is explored in the following sections.

Contents

1 Overview

All algorithms are based on the fact that the starting position does not contribute to the overal cost or score. Thus, if the start and goal are located on the same square, the cost and length would both have a value of zero.

1.1 Maps

The maps used in this project are created directly from the python program as image files. Colors are used to illustrate the type of terrain:

Color	Meaning	Cost
gray	road	1
tan	field	2
green	forest	4
light brown	hills	5
blue	river	7
dark brown	mountains	10
dark blue	water	invalid

The images have a particular legend to them as well:

Style	Meaning
solid red	path
outline red	explored (closed)
outline purple	frontier (open)
outline yellow	start
outline green	goal

2 Implementation

All of the pathfinding algorithms presented use the same base algorithm with slight modifications to on ordering in order to achieve the desired result. Shown below is peusdocode of the python implementation for all these algorithms:

```
create open and closed sets
add start to open set
while open set is not empty:
    take parent from open set
    if parent is goal:
        return path
    add parent to closed set
    for all children in expanded parent:
        if child is not in closed set:
            child cost = previous score + cost of child
            if (child is not in open set) or
               (child cost is less than previous score):
                path[index] = parent
                previous score = child cost
                new child score = child cost + heuristic applied to child
                if child is not in open set:
                    add child to open set
```

2.1 Heuristics

The two heuristics used are Manhattan and Euclidean. Shown below are how these calculations work in peusdocode:

```
manhattan(current):
    x0, y0 = goal
    x1, y1 = current
    return abs(x0 - x1) + abs(y0 - y1)

euclidean(current):
    x0, y0 = goal
    x1, y1 = current
    return sqrt(x0 - x1)^2 + (y0 - y1)^2)
```

2.2 Results

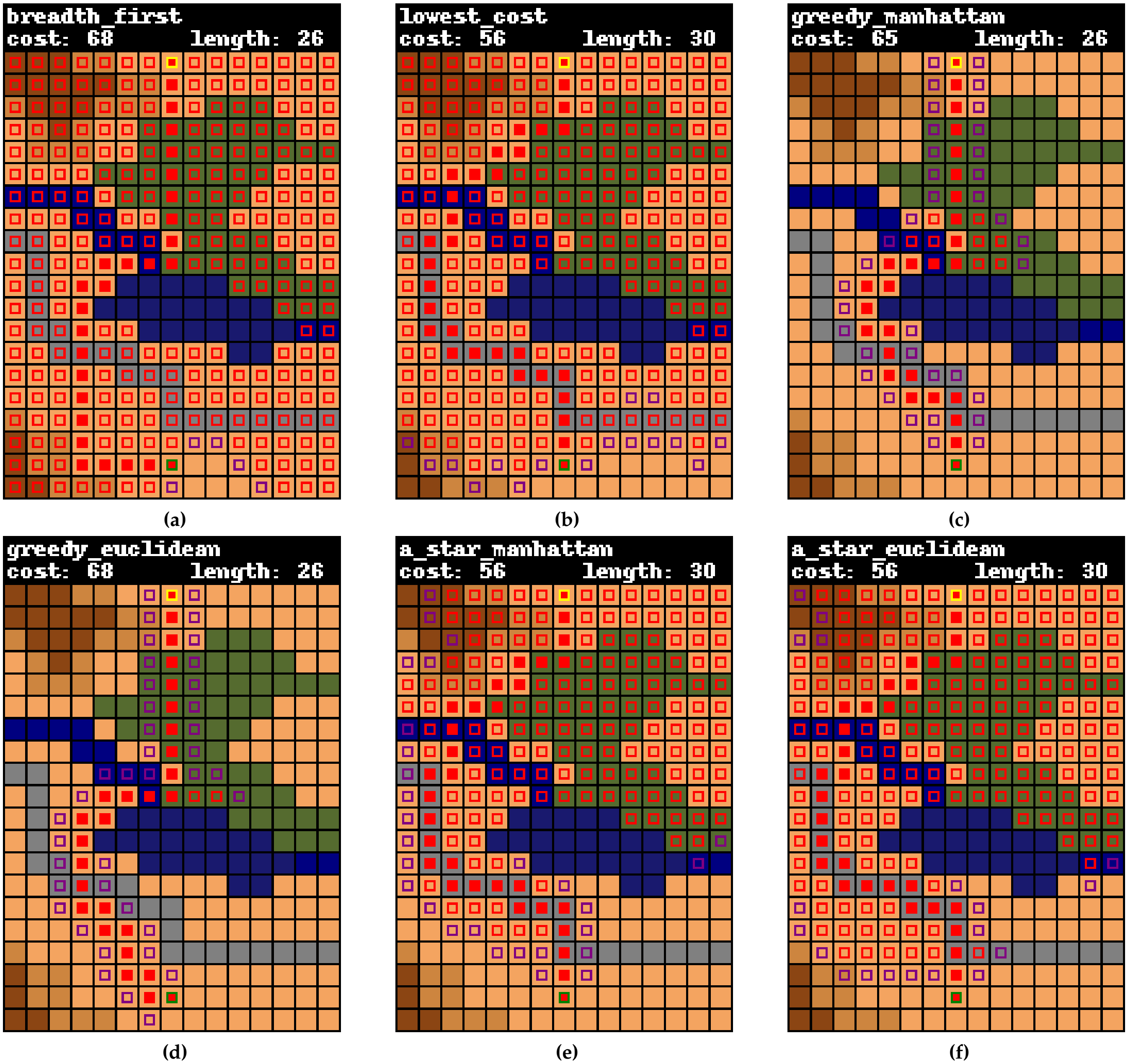

Figure 1: Various results from different search algorithms. The costs and lengths are printed directly onto the image files

2.3 Breadth-First

Breadth-first examines almost every possible path until the goal is hit. This gives it a rather worrisome time and space complexity of $O(b^{d+1})$. Starting with an initial parent, it branches out to surrounding children recursively. It therefore will always find any existing solution, and the solution with the shortest possible path. From the reference implementation, costs, scores, and heuristics are not used in the breadth-first search. Parents are taken from the open list in a first in first out (FIFO) queue style. This is shown above in Figure 1. We note that it needs to explore nearly every cell, nearly 90% of the example map.

2.4 Least Cost

The least cost algorithm provides the path of least cost, which may or may not be the shortest path. From the reference implementation, parents are taken from the open list in order of their cost, unlike breadth-first search. This still requires traversing a large portion of the map, in the example this equates to 80% exploration. It is able to quickly utilize the road though to move goods for when it decides to be come sentient and take over the world.

2.5 Greedy Best First

The greedy best first algorithm uses only heuristics to compute the path, not using the costs associated with certain terrain. This reduces the explored region to roughly 22%, which is a huge increase in speed and ability to find the goal. However, this is not without cost. Compared to the least cost implementation, it requires much more stamina, and is on par with the result obtained from the breadth-first search, albeit much faster. The two different heuristics also produce different paths, however by coincidence they are exactly the same.

2.6 A*

A* provides a balance between the various algorithms. Worst case scenario means $O(n * n)$ complexity, however this is mitigated by using the different applied heuristics. From the above results, the Euclidean heuristic needed to explore a fair bit more than the Manhattan distance. We can see that it avoided the mountain range in the upper left corner, and did not waste time with the right section of the map as does the breadth-first and lowest cost algorithms.

3 Conclusion

The above algorithms all have different desires in mind, and thus it is impossible to conclude that one is better than another. Based on the results though, A* should be used in most standard implementations as it provides a nice middle ground between time and space complexity. All in all, this project was rather fun and Python made it possible to implement all these different routines in ~200 lines which was a lot better than attempting to do this in a different language.

4 Code Appendix

```python
#!/usr/bin/env python3
#
# pathfinding algorithm implementations of:
#   * breadth first
#   * lowest cost
#   * greedy best first
#   * A*
# heuristics:
#   * manhattan / euclidean
#
# output:
#   <type>.png
#
# (c) matt waltz - spring 2018

from PIL import Image, ImageDraw


class Find(object):
    def __init__(self, filename=None, style=None, heuristic=None):
        with open(filename) as f:
            data = f.readlines()

        self.costs = {'R': 1, 'f': 2, 'F': 4, 'h': 5, 'r': 7, 'M': 10}
        self.width = int(data[0].split()[0])
        self.height = int(data[0].split()[1])
        self.start = tuple(map(int, data[1].split()))
        self.goal = tuple(map(int, data[2].split()))
        self.map = data[3:]

        self.style = getattr(self, style)
        self.heuristic = getattr(self, heuristic)
        self.closedset = set()
        self.path = dict()
        self.f = dict()
        self.g = dict()

        self.out = 0
        self.end = self.style()
        pass

    def test(self, state):
        return self.goal == state

    def cost(self, state):
        return self.costs[self.map[state[1]][state[0]]]

    def valid(self, state):
        x, y = state
        return x >= 0 and y >= 0 and x < self.width and y < self.height and self.map[y][x] != 'W'

    def expand(self, state):
        x, y = state
        result = list()
        for neighbor in ((x, y - 1), (x, y + 1), (x + 1, y), (x - 1, y)):
            if self.valid(neighbor):
```

```python
                result.append(neighbor)
        return result

    def add_fifo(self, state):
        self.openset.append(state)

    def add_set(self, state):
        self.openset.add(state)

    def take_first(self):
        self.out += 1
        return self.openset[self.out - 1]

    def take_sorted(self):
        index = 0
        fsort = self.sort()
        for index in range(len(fsort) - 1):
            if fsort[index] not in self.closedset:
                break
        send = fsort[index]
        self.openset.remove(send)
        return send

    def sort_heuristic_cost(self):
        return sorted(self.f, key=lambda state: self.g[state] + self.heuristic(state))

    def sort_heuristic(self):
        return sorted(self.f, key=lambda state: self.heuristic(state))

    def sort_cost(self):
        return sorted(self.f, key=lambda state: self.g[state])

    def breadth_first(self):
        self.openset = list()
        self.take = self.take_first
        self.add = self.add_fifo
        self.score = self.none
        return self.process()

    def lowest_cost(self):
        self.openset = set()
        self.take = self.take_sorted
        self.add = self.add_set
        self.sort = self.sort_cost
        self.score = self.cost
        return self.process()

    def greedy(self):
        self.openset = set()
        self.take = self.take_sorted
        self.add = self.add_set
        self.sort = self.sort_heuristic
        self.score = self.none
        return self.process()

    def a_star(self):
        self.openset = set()
        self.take = self.take_sorted
```

```
        self.add = self.add_set
        self.sort = self.sort_heuristic_cost
        self.score = self.cost
        return self.process()

    def process(self):
        self.g[self.start] = 0
        self.f[self.start] = self.heuristic(self.start)
        self.add(self.start)
        self.path[self.start] = None

        while len(self.openset):
            parent = self.take()
            if self.test(parent):
                return parent
            self.closedset.add(parent)
            for child in self.expand(parent):
                if child not in self.closedset:
                    child_cost = self.g[parent] + self.score(child)
                    if child not in self.openset or child_cost < self.g[child]:
                        self.path[child] = parent
                        self.g[child] = child_cost
                        self.f[child] = child_cost + self.heuristic(child)
                        if child not in self.openset:
                            self.add(child)

    @staticmethod
    def none(state):
        return 0

    def euclidean(self, state):
        x0, y0 = self.goal
        x1, y1 = state
        return ((x0 - x1) ** 2 + (y0 - y1) ** 2) ** (1.0 / 2)

    def manhattan(self, state):
        x0, y0 = self.goal
        x1, y1 = state
        return abs(x0 - x1) + abs(y0 - y1)

    def get(self):
        total = 0
        moves = [self.end]
        state = self.path[self.end]
        while state is not None:
            moves.append(state)
            total += self.cost(state)
            state = self.path[state]
        moves.reverse()
        return moves, total


def main():
    types = ['breadth_first', 'none',
             'lowest_cost', 'none',
             'greedy', 'none',
             'greedy', 'manhattan',
             'greedy', 'euclidean',
```

```
                'a_star', 'manhattan',
                'a_star', 'euclidean']

    terrain = {'R': [128, 128, 128], 'f': [244, 164, 96],
               'F': [85, 107, 47], 'h': [205, 133, 63],
               'r': [0, 0, 128], 'M': [139, 69, 19],
               'W': [25, 25, 110]}

    for j in range(0, len(types), 2):
        type_str = types[j]
        if types[j + 1] is not 'none':
            type_str += '_' + types[j + 1]

        try:
            search = Find('map.txt', types[j], types[j + 1])
            path, cost = search.get()
            length = len(path) - 1
        except Exception:
            return print('error: no path found')

        offset = 21
        width = search.width * 10 + 1
        height = search.height * 10 + 1
        img = Image.new('RGB', (width, height + offset), color='black')

        draw = ImageDraw.Draw(img)
        for x in range(search.width + 1):
            nx = x * 10
            for y in range(search.height + 1):
                ny = y * 10 + offset
                if y < search.height and x < search.width:
                    color = terrain[search.map[y][x]]
                    outline = ((nx + 3, ny + 3), (nx + 7, ny + 7))
                    solid = ((nx + 4, ny + 4), (nx + 6, ny + 6))
                    draw.rectangle(((nx + 1, ny + 1), (nx + 9, ny + 9)), tuple(color))
                    if (x, y) in path:
                        draw.rectangle(solid, 'red')
                    if (x, y) in search.closedset:
                        draw.rectangle(outline, None, 'red')
                    elif (x, y) in search.openset:
                        draw.rectangle(outline, None, 'purple')
                    if (x, y) == search.start:
                        draw.rectangle(outline, None, 'yellow')
                    elif (x, y) == search.goal:
                        draw.rectangle(outline, None, 'green')
            draw.text((2, 0), type_str, (255, 255, 255))
            draw.text((2, 10), 'cost: ' + str(cost), (255, 255, 255))
            draw.text((width - 67, 10), 'length: ' + str(length), (255, 255, 255))

        print('wrote: ' + type_str + '.png')
        print('cost: ' + str(cost))
        print('length: ' + str(length))
        print('closed length:' + str(len(search.closedset)))
        img.save(type_str + '.png')


main()
```