Capstone Project

Machine Learning Engineer Nanodegree

Sen Lyu
January 26st, 2017

Definition

Project Overview

This project is trying to find a software aspect solution for the robot to find the best way to solve a maze by itself. The project could be considered as an algorithm for the micormouse competition.

The maze is a combination of cells in the size of n*n (n will be the even number, and normally n is at least 12). The maze has a 2*2 area in the center, which is the goal for the robot to be. The robot attending whatever one cell of the goal area will be considered reach the goal. And the maze will have walls. For each cell, there could be 4 kinds of walls, upside, downside, left side, right side. The robot could not pass the walls. And we assume the maze always has a solution to the goal area.

The robot will start at the left bottom of the maze. We don't consider the size of the robot, we assume it won't affect the walking of the robot. The robot could move for four directions using same time. And because the robot needs to learn the maze, we assume when the robot come to a cell, it will learn all the information of the maze, which means, in details, the walls of the cell won't be learn until it enters the cell.

And the search for the maze will be in two phase. At first, the robot will search the maze until it reaches the goal area, and it could keep searching. If it considered done searching, it must go back to the start point. All this consider the first phase. In the second phase, the robot will go to the goal area again and come back. The time in both phase will be used to determine the score of a robot search.

Problem Statement

We will have the information of the maze as input. The problem we are facing in this project is how to find a way to solve the maze in the shortest time. And because the robot won't have the information of the cell until it attends the cell, so it will be a graph search. And it needs to do it in the shortest time. So it will also needs an algorithm to find the shortest way between two points.

Metrics

The evaluation of the model is simple, we just focus on the time cost. If a model has a less time cost, it will be the better one. And if we assume the first step time cost to be C_1, the second step time cost to be C_2, the total time cost function will be:

$$\mathrm{TC} = \frac{1}{30} C_1 + C_2$$

Analysis

(approximately 2 - 4 pages)

Data Exploration

The input of the data will be in two parts. The first part is one number n as the length of the maze. The second parts are a n*n array which is the walls information of each cell. An example in Chart 1.

```
1,5,7,5,5,5,7,5,7,5,5,6
3,5,14,3,7,5,15,4,9,5,7,12
11,6,10,10,9,7,13,6,3,5,13,4
10,9,13,12,3,13,5,12,9,5,7,6
9,5,6,3,15,5,5,7,7,4,10,10
3,5,15,14,10,3,6,10,11,6,10,10
9,7,12,11,12,9,14,9,14,11,13,14
3,13,5,12,2,3,13,6,9,14,3,14
11,4,1,7,15,13,7,13,6,9,14,10
11,5,6,10,9,7,13,5,15,7,14,8
11,5,12,10,2,9,5,6,10,8,9,6
9,5,5,13,13,5,5,12,9,5,5,12
```

Chart 1

The number will be in the set of [0,15], each number represents a four-bit number that have a bit value of 0 if this side have no wall, 1 if wall, in the sequence of up, right, bottom, left. For example, if the number is 1, it will have 3 walls in left right and bottom. And if we change the data to the picture it will be like Chart 2.

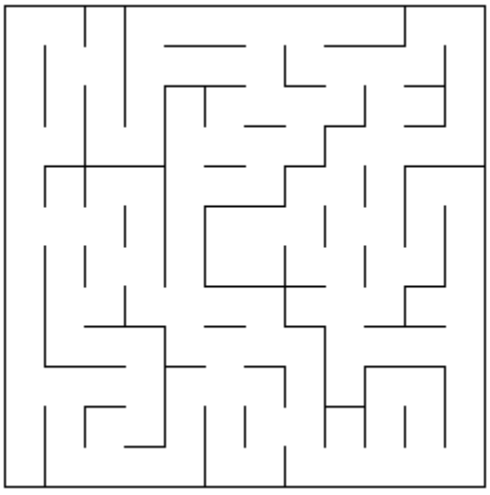


Chart 2

And in order to use the graph algorithm, I need to use the array V[i,j] to store the points (i,j).

$$\mathrm{V[i,j]} = \begin{cases} 0 & if\ the\ point\ is\ not\ visied \\ 1 & if\ the\ point's\ near\ pont\ is\ visited \\ 2 & if\ the\ point\ is\ visied \end{cases}$$

$$\mathrm{i,j} \in [0,\mathrm{n}]\ (n\ is\ the\ length\ of\ the\ maze)$$

And also I need a G[a,b] to store all the edges (a, b both are points).

$$\mathrm{G[a,b]} = \begin{cases} 1 & if\ a\ and\ b\ are\ connected \\ 9999 & if\ a\ and\ b\ are\ not\ connected \end{cases}$$

$$\mathrm{a,b} \in [0,\mathrm{n} * \mathrm{n}]\ (n\ is\ the\ length\ of\ the\ maze)$$

Algorithms and Techniques

There are three main algorithms in my project. First is graph searching, which is used for robot walking in the maze. Second is Dijkstra Algorithm for solve the single source shortest path. And the third one is used to determine whether the robot has already find the shortest way to walk out the maze.

The graph search I used here is the depth first search. Because the breadth first search will make the robot run back and force it is not good. And the depth first search is better here.

The Dijkstra Algorithm is good for the single source shortest path here.

The algorithm of finding whether the robot is already find the shortest way is doing this. Let's assume that we are at a state that the robot is searching in the maze, now there are some points it attends and know the walls information of the cells. But there are still some points it hasn't searched. So, we can assume the points which the robot hasn't searched have no wall. Then we find the shortest way from the start to the goal. If the shortest way doesn't contain the points that the robot hasn't searched, then we have the shortest path of all maps.

Benchmark

The benchmark of this solution is about the way of search. Because when the robot does the search, instead it wants to go deeper, it wants to go to the closest point to the goal. But before the search we no idea how close a point is to the goal area. So a simple way to do this is searching for the point whose coordination is near to the goal area, which means we want to find the point:

$$\min(\mathrm{abs}(\mathrm{x} - \frac{\mathrm{dim}}{2} - 0.5, \mathrm{asb}(\mathrm{y} - \frac{\mathrm{dim}}{2} - 0.5)))$$

Methodology

(approximately 3 - 5 pages)

Data Preprocessing

Because we need points and edges for both searching algorithm and the shortest path algorithm, we need to change the inputs to points array and the edges array. So the most work here is to transform the input information into the edges array.

The first thing I do is change the (i,j) coordination of the cells to a number, so that the edge array could be a two dimensional array.

```
def change_point(a,b):
    point = a * dim + b
    return point

def change_back_point(point):
    a = point / dim
    b = point % dim
    return a,b
```

The transform is look like this:

```
for i in range(dim):
    for j in range(dim):
        x = map_info[i,j]
        up, right, botton, left = False, False, False, False
        if x >= 8:
            left = True
            x -= 8
        if x >= 4:
            botton = True
            x -= 4
        if x >= 2:
            right = True
            x -= 2
        if x >= 1:
            up = True
        if (up and j+1 < dim):
            E[change_point(i,j),change_point(i,j+1)] = 1
            E[change_point(i,j+1),change_point(i,j)] = 1
        if (right and i+1 < dim):
            E[change_point(i,j),change_point(i+1,j)] = 1
            E[change_point(i+1,j),change_point(i,j)] = 1
        if (botton and j-1 >= 0):
            E[change_point(i,j),change_point(i,j-1)] = 1
            E[change_point(i,j-1),change_point(i,j)] = 1
        if (left and i-1 >= 0):
            E[change_point(i,j),change_point(i-1,j)] = 1
            E[change_point(i-1,j),change_point(i,j)] = 1
```

So after all this data preprocess, we can use the algorithm to do the rest of the work.

Implementation

The whole process is very simple, the robot does depth first walking until it finds the goal, then calculate whether it finds the shortest way, if it does then stop search and go back to the start point, if not then keep searching. So here are two things the robots will do repeat, one is walk, the second is calculate whether it finds the shortest way.

The walk is according the depth first search. First set all the points near now to be the points accessible (in the code is V[i,j] = 1). Then finds the nearest point to the now point, set it to be the next. And then walk to the next point.

```
def run(E,V,now,prev,time):
    nextp = now
    for i in range(E.shape[1]):
        if E[now,i] == 1:
            x,y = change_back_point(i)
            if V[x,y] == 0:
                V[x,y] = 1
    mindis = 9999
    prelistmin = []
    for i in range(dim):
        for j in range(dim):
            if V[i,j] == 1:
                V_new, E_new, start_new, end_new, getback = know_area(V,E,now,change_point(i,j))
                greydis, prelist = shortest(V_new, E_new, start_new, end_new, getback)
                if greydis < mindis:
                    mindis = greydis
                    prelistmin = prelist
                    nextp = change_point(i,j)
```

The find of the shortest way is a little bit tricky here. Because as the robot we assume is searching the maze, it doesn't have all the points of the maze, so when we do the shortest path calculation, we need a subset of both the points and the edges.

```
def know_area(V,E,start,end):
    k = 0
    V_new = []
    getback = []
    for i in range(V.shape[0]):
        for j in range(V.shape[0]):
            if V[i,j] != 0:
                V_new.append(change_point(i,j))
                if change_point(i,j) == start:
                    start_new = k
                elif change_point(i,j) == end:
                    end_new = k
                getback.append(change_point(i,j))
                k += 1
    E_new = np.ndarray(shape=(len(V_new),len(V_new)),dtype=np.int32)
    E_new.fill(9999)
    for i in range(len(V_new)):
        for j in range(len(V_new)):
            if E[V_new[i],V_new[j]] == 1:
                E_new[i,j] = 1
    return V_new, E_new, start_new, end_new, getback
```

After get the subset of the maze, we can now do the Dijkstra Algorithm

```
def shortest(V,E,start,end,getback):
    w = [ 999 for i in range(len(V))]
    w[start] = 0
    pre = [ 0 for i in range(len(V))]
    Q = set( i for i in range(len(V)))
    while len(Q)>0:
        minn = 9999
        for element in Q:
            if w[element]<minn:
                minn, u = w[element], element
        Q.remove(u)
        for i in range(E.shape[0]):
            if E[u,i] == 1:
                alt = w[u]+1
                if alt < w[i]:
                    w[i], pre[i] = alt, u
    def printout(start,end,prelist):
        if start == end:
            pass
        else:
            new_end=pre[end]
            prelist = printout(start,new_end,prelist)
            prelist.append(end)
        return prelist
    prelist = []
    prelist = printout(start, end, prelist)
    prev = []
    for i in range(len(prelist)):
        prev.append(getback[prelist[i]])
    return w[end], prev
```

So the function to decide whether the robot is found the shortest way:

```
def finded(E,V):
    V_all = V.copy()
    V_all.fill(2)
    V_new, E_new, start_new, end_new, getback =
know_area(V_all,E,change_point(0,0),change_point(dim/2,dim/2))
    short2, short2list = shortest(V_new, E_new, start_new, end_new, getback)
    flag = True
    for i in range(len(short2list)):
        a,b=change_back_point(short2list[i])
        if V[a,b] != 2:
            flag = False
            break
    return flag
```

Refinement

Because at first the depth first search will choose the direction of a sequences “left, down, up, right”. This was because the loop variable is from the left to right and down to up.

At first the results are like this:

	solution				
	time1	time2	time3	time4	score
test1	104	0	30	60	64.467
test2	183	34	21	88	95.933
test3	271	79	40	100	113

The solution is good for the first two, but the third one is worse than the benchmark. And because the benchmark is going according the ordination, so I was thinking instead using a sequence of “left, down, up, right”, why not make the model more aggressive? When the model needs to decided, which direction should choose when the others are the same, why not let the model choose the nearest one according to the ordination?

So I added that and the results turns to be like this:

Although, the results seem not much improved, and from the first two the model seems compromised. But introduce greedy to the algorithm seem will make the algorithm improved a lot in the special situation.

Results

(approximately 2 - 3 pages)

Model Evaluation and Validation

	solve improve					
	time1	time2	time3	time4	score	Shortest way
test1	138	51	13	60	66.733	30
test2	137	134	5	88	97.2	44
test3	235	107	22	100	112.133	50

The results of the solution are reasonable. The solution is solid and stable. And the results can be trusted.

Justification

The solution is better than the benchmark a lot in the first two cases. But a little behind in the third case.

	solve improve					benchmark			
	time1	time2	time3	time4	score	time1	time3	time4	score
test1	138	51	13	60	66.733	596	30	60	80.867
test2	137	134	5	88	97.2	591	45	92	113.2
test3	235	107	22	100	112.133	177	51	104	111.6

Conclusion

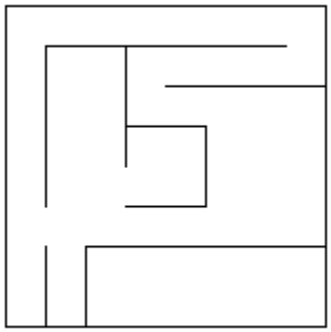
(approximately 1 - 2 pages)

Free-Form Visualization

Here I will try to display the situation which prove the improved model in some special situation will bring big improvement to the solution.

I designed a small maze to find why the benchmark could do better than my solution.

The maze looks like this:



The banch mark goes:

('score', 16.533333333333335)
(0, 1)
(0, 2)
(0, 3)
(0, 4)
(0, 3)
(0, 2)
(1, 2)
(1, 3)

(2, 3)
(3, 3)

The older model goes:

('score', 17.6)
(0, 1)
(0, 2)
(0, 3)
(0, 4)
(0, 5)
(0, 6)
(0, 7)
(1, 7)
(2, 7)
(3, 7)
(4, 7)
(5, 7)
(6, 7)
(7, 7)
(7, 6)
(6, 6)
(5, 6)
(4, 6)
(3, 6)
(3, 5)
(4, 5)
(5, 5)
(5, 4)
(5, 3)
(5, 2)
(4, 2)
(3, 2)
(2, 2)
(1, 2)
(1, 1)
(1, 0)
(1, 1)
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
(2, 6)
(2, 5)
(2, 4)
(2, 3)
(3, 3)

The new model goes:

('score', 16.4)
(0, 1)
(0, 2)
(1, 2)
(1, 3)
(2, 3)
(3, 3)

The new model goes directly to the goal area.

Reflection

The biggest problem of my model is that there is too much calculation to do. The code running slowly. Because in the search part, I use the shortest path to determined which path should be reach next. And the shortest path algorithm cost a lot of time. So in realism, this will make the robot search slower.

Improvement

If we think the scenario in the continuous base, which means:

1) If there is no wall, the robot's sense could find more nearby cells' information from one point.

2) The robot will have speed and accelerate speed in the moving, which will change the distance measured in time of the nearby cells, which means if the robot doesn't have to move, it should go straight.

3) The turning of the robot will cost more time than go straight, so in some case, it doesn't have to take turns, it could just go straight, which will be faster.