Term Project on Application of Genetic Algorithm

**Topic: Path Planning using Genetic Algorithms**

Course: MT21104 -**Genetic Algorithms in Engineering Process Modeling**

Abstract:

This term project aims to illustrate the use of Simple Genetic Algorithm(SGA) for Path Planning in Mobile Robots. Although there are several papers on usage of fuzzy logic and evolutionary neural networks in this realm, it is always evident that simple and clean turns out to be highly efficient. Hence the paper(1) was selected for reference owing to it's simplicity and not surprisingly, our own implementation for the same from the scratch proved to be satisfactory. The SGA is used to find optimal/near-optimal path for a robot in a static map with nodes as path points and edges as feasible steps. The robot is assumed to traverse in a 2D space represented as a grid. Each feasible location on the grid(real co-ordinate) is represented as a binary encoding. The Robot has a starting point and a target point which it travels along the via points if they are feasible in such a way that the total path length is minimum.

Introduction:

The Path Planning Problem in brief can be Presented as => Given data: (1. Starting Location 2. Goal Location 3. 2-D Workspace consisting node locations and connectivity), Plan a collision free trajectory between two specified points satisfying criteria for optimal path or shortest path. The Path Planning Problem is computationally very expensive. In spite of a great research being done to address this problem, conventional approaches become infeasible due to

* Single solution improvement strategy.
* Limits of Computational Resources(memory).

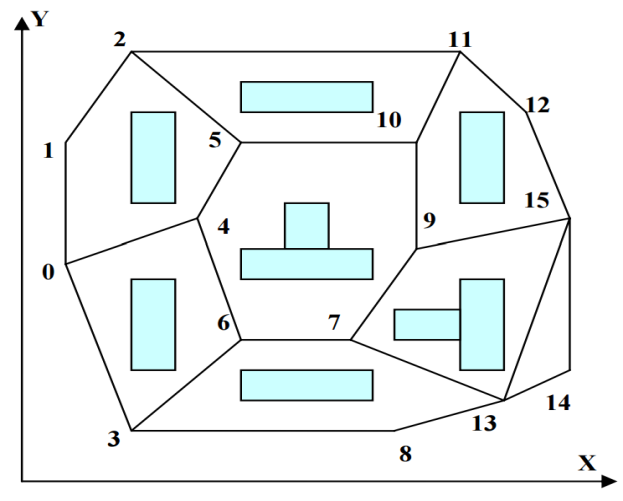
Hence by using Genetic Algorithm and utilizing it's multiple solution search, we have been able to present a solution which is both flexible and scalable.

Problem Formulation:

The Area Described by 2-D Workspace is the region Allowed for mobile robot to move. This 2-D map includes Several obstacles which are Provided at Runtime. These obstacles are represented by set of points which allows the mobile robot to avoid them. The Starting and Destination points are also Provided. The main Objective is to plan a collision free path which is also the optimal(shortest) path between the source and destination. The following design objectives are considered while finding the optimal path :-

* Minimize Distance travelled Between Source and Destination.
* Travel through one point only once.
* Reject Infeasible paths

In solving this problem we have assumed that the map of the location with the position and size of the obstacles has already been calculated and provided to us in the form of points through which the robot can move. The points and the interconnecting paths specify the permissible paths for the mobile robot .



Mapping the Workspace to find permissible points/paths

Procedure and Algorithm :

As stated earlier the code used takes the position(co-ordinates) and connectivity of the points as inputs . After which one can specify the Start and End points upon which code will specify the shortest feasible path between those two points.

First we started off by taking the co-ordinates and connectivity of the nodes as input . These were stored for easy access in subsequent steps . Then the code generates a random population of strings specifying the path co-ordinates and assigns fitness to the individuals.

Fitness is defined as 1/Path Distance for feasible paths and 0 for infeasible paths .

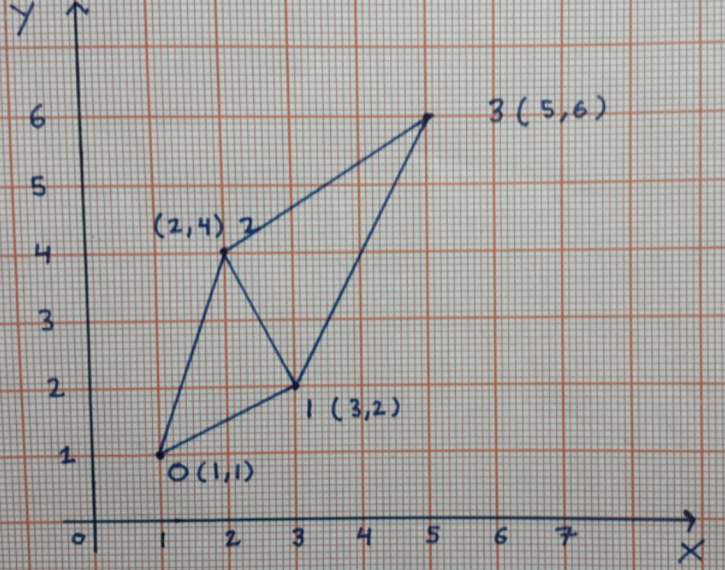
Selection and Crossover is carried on the current generation to find the next generation . We didn't consider mutation while finding the next generation .The Selection operator considered is quite simple , wherein we drop the least fit individual and make 2 copies of the fittest individual . We have considered single point crossover while evaluating the next generation.

The code is written in python and has been included in a separate document for reference .

Simulation and Results :-

We did some simulations for some simple cases with limited number of nodes .

1.



Number of Nodes : 4 (0,1,2,3)

Starting Node : 0

Ending Node : 3

Population after 1000 iterations :

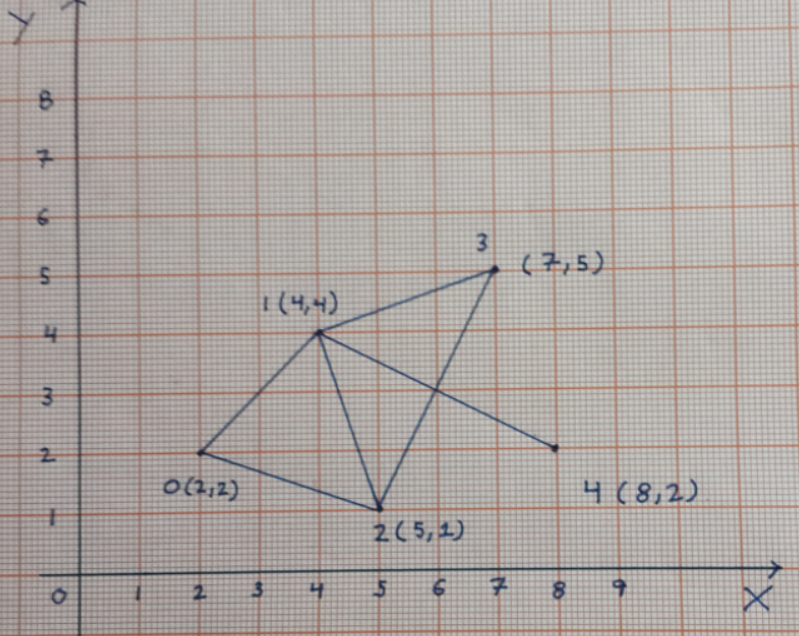
[['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1'], ['0', '0', '0', '0', '0', '1', '1', '1']]

Best: (-0.14907119849998599, ['0', '0', '0', '0', '0', '1', '1', '1'])

Final Solution : ['0', '0', '0', '0', '0', '1', '1', '1']

**Solution : 0 ---> 1 ---> 3**

2.

****

Number of Nodes : 5 (0,1,2,3,4)

Starting Node : 0

Ending Node : 4

Population after 1000 iterations(Population Size 40) :

[['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'], ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0']]

Best: (-0.1369757358544491, ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0'])

Final Solution : ['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '0', '0']

**Solution : 0--->1--->4**

Limitations and possible improvements:

* In our implementation, for simplicity, we have considered only finite nodes as feasible steps that are allowed and instead of considering obstacle locations have included the possible transition steps that do not collide as a user defined connectivity between the nodes.
* Also currently, the convergence criteria is a a fixed number of iterations that has been determined by multiple trial-runs on the code and might lead to inefficiency.
* A very trivial selection and cross-over operator is used and can be improved. A 2-Point Crossover instead of single point Crossover has been done and the best individual is selected at each generation.
* No mutation operator has been used. This was done knowingly as mutation might cause converging solutions to wander away.
* This code presents a basic framework of working representation of the GA implementation of path planning which can be used for further works ahead.
* The major advantage of this code is modularity which allows further improvement in individual modules without disturbing the rest.
* This can be further improved using some image processing for an actual robot and produce output on a larger 2-D grid(Appendix 2).

Conclusion :

While implementing this code, we could really feel the essence of encoding a solution. The binary encoding used could help simplify the representation a lot. We have allowed user-specified number of nodes and calculated a generic size of this encoding that would be efficient and will vary with number of nodes. We used to two mapping to switch between different representations of the solution as and when required. The simplicity of the concept and it's astoundingly accurate results has really encouraged the development of more such algorithms and their improvements.

Appendix:(Code in Python)

### CODE ####

import sys

sys.path.append('/usr/lib/python2.7/dist-packages')

import numpy as np

#import cv2

import Queue

import heapq

import math

#Function to get n bit binary representation of integer x

get\_bin = lambda x, n: format(x, 'b').zfill(n)

#Function to real representation of a binary numer fed in the form of a list

def get\_real\_list(x):

real = 0

for i in range(len(x)):

real +=x[i]\*2\*\*(len(x)-i-1)

return real

#Number of nodes or path points to be travelled: nodes

nodes = int(input("Enter number of nodes: "))

#Size of encoding of a single string representing a node: enc\_size

enc\_size = int(math.ceil(math.log(nodes,2)))

#Maps

map\_1 = {} #Maps binary to co-ordinates

map\_2 = {} #Maps nodenumbers to binary

#Encoding of all the nodes into binary

print "Enter the co-ordinates(x y) for all the nodes"

for i in range(nodes):

print "node ("+str(i)+") : "

a = int(input("x: "))

b = int(input("y: "))

bin = get\_bin(i,enc\_size)

map\_1[bin] = (a,b)

map\_2[i] = bin

#New code snippet to enter starting and ending nodes

print "Nodes List \n"

print range(nodes)

print

print "Please enter only those nodes in the node list \n"

start = int(input("Enter the starting node : "))

end = int(input("Enter the ending node : "))

print

print

#Enter the connectivity of the nodes

nodes\_conn = {}

for i in range(nodes):

a = map\_1[get\_bin(i , enc\_size)]

ch =int(input("Number of nodes connected to Node "+str(i)+" : "))

print "Enter the nodes connected to Node "+str(i)+" : "

#ch = "Y"

nodes\_conn[a] = [a]

for ch in range(ch):

temp = int(input("Enter the neighbouring node :"))

temp = map\_1[get\_bin(temp , enc\_size)]

nodes\_conn[a].append(temp)

#String and population size computation

print

str\_size = nodes\*enc\_size #change this to no.of.obstacles \* enc\_size

pop\_size = int(input("Enter number of individuals in a population: "))

print

#Generate initial population

pop = []

for i in range(pop\_size):

ind = []

for x in range(len(map\_2[start])):

ind.append(int(map\_2[start][x]))

mid\_size = str\_size - 2\*enc\_size

for j in range(mid\_size/enc\_size):

while(1):

node\_allowed = list((np.random.choice([0,1], size=(enc\_size,))))

if (get\_real\_list(node\_allowed)<nodes):

break

ind+=node\_allowed

for x in range(len(map\_2[end])):

ind.append(int(map\_2[end][x]))

print "Ind"+str(i)+" "+str(ind)

pop.append(ind)

print "Population: "

print pop

print

print

#Initial Fitnesss

fit = [0 for x in range(pop\_size)]

#Calculating the fitness of the population

def calc\_fitness(pop, fit):

#Calculating the fitness of the populaton,calls another function that calculates the indivisual population

#Assign fitness considering connectivity and euler distance between two nodes

for i in range(pop\_size):

fit[i] = calc\_fitness\_indivisual(pop[i])

def calc\_fitness\_indivisual(indi):

#converting to a list of characters to make it simple

for i in range(len(indi)):

indi[i] = str(indi[i])

total\_dist = 0

for i in range(len(indi)/enc\_size - 1):

current\_node\_coordinates = map\_1["".join(indi[i\*enc\_size:((i+1)\*enc\_size)])]

next\_node\_coordinates = map\_1["".join(indi[((i+1)\*enc\_size):((i+2)\*enc\_size)])]

if (next\_node\_coordinates in nodes\_conn[current\_node\_coordinates]):

temp\_1 = current\_node\_coordinates

temp\_2 = next\_node\_coordinates

dist = math.sqrt((temp\_2[0] - temp\_1[0])\*\*2 + (temp\_2[1] - temp\_1[1])\*\*2)

total\_dist +=dist

else:

return 0

if(total\_dist):

indi\_fit = 1/total\_dist

return indi\_fit

else:

return 0

##Iterate till convergence

convergence = False

#Priority Queue to decide the rank on the basis of fitness: pq

pq = []

calc\_fitness(pop,fit)

#Arrange on the basis of fitness

for i in range(pop\_size):

heapq.heappush(pq,(-fit[i],pop[i]))

#One with highest fitness, automatically at the end of priority queue

ctr = 0

print "Generation "+str(ctr)+": "

best\_sol = heapq.heappop(pq)

print "Best: "+str(best\_sol)

while(not convergence):

#Selection operation

best\_ind = best\_sol[1] #Check for [1] in case of error

print "Best\_ind: "+str(best\_ind)

new\_pop = []

new\_pop.append(best\_ind)

new\_pop.append(best\_ind)

for x in range(pop\_size-2):

new\_pop.append(heapq.heappop(pq)[1])

print "Reject: "+ str(heapq.heappop(pq))

print "New pop: "+str(new\_pop)

#Cross-over operation

cross\_pop = []

for x in range(pop\_size):

temp\_num = len(new\_pop[x])/2 - (len(new\_pop[x])/2)%enc\_size

cross\_pop.append(new\_pop[x][:temp\_num]+new\_pop[pop\_size-x-1][temp\_num:])

print "cross\_pop: "+str(cross\_pop)

#Iterate

ctr+=1

print "Generation "+str(ctr)+": "

calc\_fitness(cross\_pop,fit)

#Arrange on the basis of fitness

for i in range(pop\_size):

heapq.heappush(pq,(-fit[i],cross\_pop[i]))

best\_sol = heapq.heappop(pq)

print "Best: "+str(best\_sol)

if(ctr == 1000):

convergence = True

#Solution: best fit individual

print

print

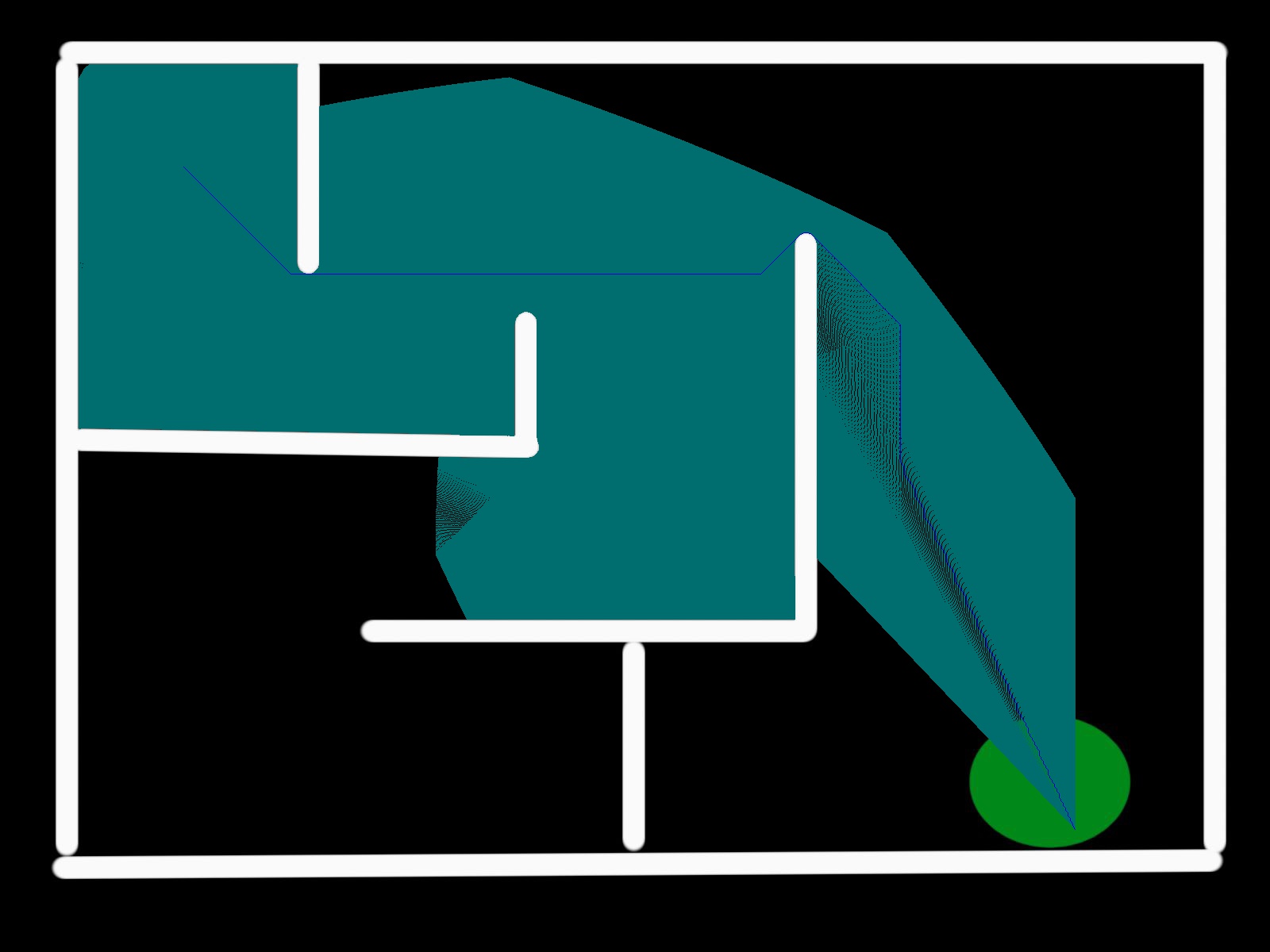
print "Final Solution: "

print best\_ind

print

print "End"

Appendix -2:

Conventional A-Star implementation by us.

Green: Target

Deep blue: Path

Light blue: Explored path

Source is at start of path.

References :

*1. Gihan NAGIB and W. GHARIEB "Path Planning for a mobile robot using genetic algorithm"*