UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO E AUTOMAÇÃO
DISCIPLINA DE INTELIGÊNCIA ARTIFICIAL APLICADA

RELATÓRIO DO PROJETO 3

Alunos: Eric Calasans de Barros
José Genilson da Silva Filho

Professor: Marcelo Fernandes

Sumário

Lista de Figuras

Lista de Tabelas

1. Introdução

Os **Algoritmos Genéticos(GA)** fazem parte do conjunto de algoritmos conhecidos como **Algoritmos Evolucionários**, que se inspiram nos mecanismos naturais de seleção e evolução das espécies para estabelecer seu paradigma. Em linhas gerais, os *GA*'s utilizam-se de operadores como **aptidão(fitness)**, **seleção**, **cruzamento(crossover)** e **mutação** para gerar a(s) melhor(es) solução(ões) para os problemas a que se propõem.

O objetivo do projeto desta unidade é utilizar os *GA*'s em um problema conhecido como **path planning**, que consiste em achar um caminho, se possível o melhor, dado um ponto de partida e um ponto de chegada e avaliar sua eficiência e eficácia na prática.

2. Materiais e Métodos

2.1. Ambiente de Desenvolvimento

Para simular o robô foi usado o ambiente de simulação da Coppelia Robotics(`http://www.coppeliarobotics.com`) conhecido como **Virtual Robot Experimentation Plataform**(VREP), dadas as dificuldades que tivemos durante o semestre para utilizar a plataforma **iRobot** associada ao MATLAB e o sistema operacional Mac OS Sierra(na ocasião encontramos problemas com a GUI do iRobot). Para desenvolvimento dos scritps optamos pela linguagens **Python** para desenvolvimento do *GA* e comando do VREP e LUA, do lado do ambiente do VREP, para fornecer os dados de entrada para os scripts Python.

2.2. Simulação do Ambiente

O ambiente de simulação foi modelado como um quadrado 5x5 onde inserimos 6 obstáculos cubóides de lado 1, dispostos conforme a tabela abaixo:

Obstáculo	centro(x,y)
obs1	(1.0,1.0)
obs2	(1.5,3.5)
obs3	(2.5,2.5)
obs4	(4.5,2.0)
obs5	(3.0,1.0)
obs6	(4.0,3.0)

Tabela 1: Coordenadas dos Obstáculos

Como objeto de simulação do robô foi utilizado o modelo **Pioneer_p3dx** disponibilizado no VREP, posicionado em (2.0,0.5). Dois objetos do tipo **dummy**, nativo do VREP, foram colocado no mapa para simbolizarem, respectivamente, a referência do sistema de coordenadas(***ref***) e o destino(***target***), este colocado em (4.5,4.0). A figura abaixo ilustra o ambiente de desenvolvimento VREP com suas configurações iniciais:

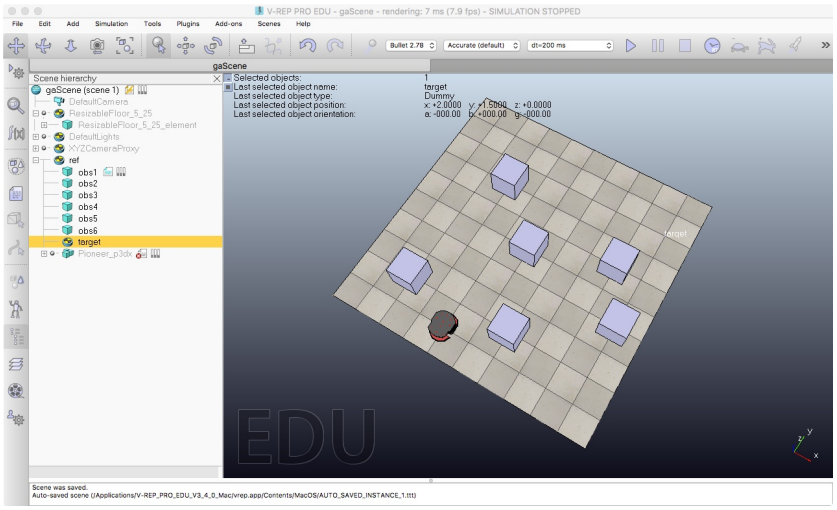

Figura 1: Configurações iniciais do VREP

Desevolvemos uma biblioteca de funções chamada `obstaculos.py` para gerar dados necessários a serem passados para o *GA*. As prinicipais funções desta biblioteca são mostradas abaixo:

- **Definição dos obstáculos** - no código abaixo são definidos os obstáculos em termos de seus 4 lados, estabelecendo cada lado como uma sequência de 20 pontos igualmente espaçados, de acordo com sua localização no mapa.

```
import numpy as np
import scipy.spatial.distance as ssd

#obstaculo 1
x1 = np.linspace(0.75, 1.25, num=20)
y1 = np.linspace(0.75, 1.25, num=20)
l1 = []
l2 = []
l3 = []
l4 = []
for i in x1:
  l1.append([i, 0.75])
  l3.append([i, 1.25])

for j in y1:
  l2.append([1.25, j])
  l4.append([0.75, j])

OBS1 = [l1, l2, l3, l4]


```

```
#obstaculo 2
x1 = np.linspace(1.25, 1.75, num=20)
y1 = np.linspace(3.25, 3.75, num=20)
l1 = []
l2 = []
l3 = []
l4 = []

for i in x1:
  l1.append([i, 3.25])
  l3.append([i, 3.75])

for j in y1:
  l2.append([1.75, j])
  l4.append([1.25, j])

OBS2 = [l1, l2, l3, l4]


#obstaculo 3
x1 = np.linspace(2.25, 2.75, num=20)
y1 = np.linspace(2.25, 2.75, num=20)
l1 = []
l2 = []
l3 = []
l4 = []

for i in x1:
  l1.append([i, 2.25])
  l3.append([i, 2.75])

for j in y1:
  l2.append([2.75, j])
  l4.append([2.25, j])

OBS3 = [l1, l2, l3, l4]

#obstaculo 4
x1 = np.linspace(4.25, 4.75, num=20)
y1 = np.linspace(1.75, 2.25, num=20)
l1 = []
l2 = []
l3 = []
l4 = []

for i in x1:
  l1.append([i, 1.75])
  l3.append([i, 2.25])

```

```
71 for j in y1:
72   l2.append([4.25, j])
73   l4.append([4.75, j])
74
75 OBS4 = [l1, l2, l3, l4]
76
77
78 #obstaculo 5
79 x1 = np.linspace(2.75, 3.25, num=20)
80 y1 = np.linspace(0.75, 1.25, num=20)
81 l1 = []
82 l2 = []
83 l3 = []
84 l4 = []
85
86 for i in x1:
87   l1.append([i, 0.75])
88   l3.append([i, 1.25])
89
90 for j in y1:
91   l2.append([2.75, j])
92   l4.append([3.25, j])
93
94 OBS5 = [l1, l2, l3, l4]
95
96 #obstaculo 6
97 x1 = np.linspace(3.75, 4.25, num=20)
98 y1 = np.linspace(2.75, 3.25, num=20)
99 l1 = []
100 l2 = []
101 l3 = []
102 l4 = []
103
104 for i in x1:
105   l1.append([i, 2.75])
106   l3.append([i, 3.25])
107
108 for j in y1:
109   l2.append([3.75, j])
110   l4.append([4.25, j])
111
112 OBS6 = [l1, l2, l3, l4]
113
```

Listing 1: Definição dos Obstáculos

- **Distância entre pontos** - função auxiliar que calcula a distância entre dois pontos através da **norma euclidiana**:

```
def distanciaPontos(point, target):
  return np.linalg.norm(target - point)

```

Listing 2: Função distanciaPontos

- **Menor distância ao obstáculo** - calcula a menor distância do ponto em questão ao lado do obstáculo mais próximo, analisando todos os obstáculos e determinando qual o mais próximo ao referido ponto(**point**), passado como parâmetro para a função:

```
def distanciasPO(point):
  distMin = []  #Matriz de distancias minimoas
  indices = []  #Matriz de indices para o ponto no obstaculo
    onde tem distMin
  temp = []  #Indices temporarios
  tempDist = []  #Distancias temporarias


  l = 0  #Inicializa o lado do obstaculo

  for lado in OBS1:  #Para um lado no obstaculo
    l += 1 #Lado atual
    distancias = [] #Matriz de distancias para um lado do
    obstaculo
    for ponto in lado:   #Para um ponto no lado atual
      d = round(np.linalg.norm(np.array(point) - np.array(
    ponto)), 2)  #Calcula a distancia do ponto ao lado
    distancias.append(d)   #Inclui a distancia na matriz
    distancias
    minDist = min(distancias)   #Calcula a menor das
    distancias do lado atual
    temp.append([l, distancias.index(minDist)])   #Inclui o
    lado e o indice do ponto da menor distancia
    tempDist.append(minDist)   #Inclui a menor distancia
    para o lado atual
  minTempDist = min(tempDist)  #Calcula a menor distancia
    entre os lados do obstaculo
  distMin.append(minTempDist)  #Insere a menor distancia do
    obstaculo
  indices.append(temp[tempDist.index(minTempDist)])  #Inclui
    o indice do lado e a coordenada do ponto da menor
    distancia
  temp = []
  tempDist = []

  l = 0
  for lado in OBS2:
    l += 1
```

```
    distancias = []
    for ponto in lado:
      d = round(np.linalg.norm(np.array(point) - np.array(
  ponto)), 2)
      distancias.append(d)
    minDist = min(distancias)
    temp.append([l, distancias.index(minDist)])
    tempDist.append(minDist)
  minTempDist = min(tempDist)
  distMin.append(minTempDist)
  indices.append(temp[tempDist.index(minTempDist)])
  temp = []
  tempDist = []

  l = 0
  for lado in OBS3:
    l += 1
    distancias = []
    for ponto in lado:
      d = round(np.linalg.norm(np.array(point) - np.array(
  ponto)), 2)
      distancias.append(d)
    minDist = min(distancias)
    temp.append([l, distancias.index(minDist)])
    tempDist.append(minDist)
  minTempDist = min(tempDist)
  distMin.append(minTempDist)
  indices.append(temp[tempDist.index(minTempDist)])
  temp = []
  tempDist = []

  l = 0
  for lado in OBS4:
    l += 1
    distancias = []
    for ponto in lado:
      d = round(np.linalg.norm(np.array(point) - np.array(
  ponto)), 2)
      distancias.append(d)
    minDist = min(distancias)
    temp.append([l, distancias.index(minDist)])
    tempDist.append(minDist)
  minTempDist = min(tempDist)
  distMin.append(minTempDist)
  indices.append(temp[tempDist.index(minTempDist)])
  temp = []
  tempDist = []

  l = 0
```

```
74    for lado in OBS5:
75      l += 1
76      distancias = []
77      for ponto in lado:
78        d = round(np.linalg.norm(np.array(point) - np.array(
      ponto)), 2)
79        distancias.append(d)
80      minDist = min(distancias)
81      temp.append([l, distancias.index(minDist)])
82      tempDist.append(minDist)
83    minTempDist = min(tempDist)
84    distMin.append(minTempDist)
85    indices.append(temp[tempDist.index(minTempDist)])
86    temp = []
87    tempDist = []
88
89    l = 0
90    for lado in OBS6:
91      l += 1
92      distancias = []
93      for ponto in lado:
94        d = round(np.linalg.norm(np.array(point) - np.array(
      ponto)), 2)
95        distancias.append(d)
96      minDist = min(distancias)
97      temp.append([l, distancias.index(minDist)])
98      tempDist.append(minDist)
99    minTempDist = min(tempDist)
100   distMin.append(minTempDist)
101   indices.append(temp[tempDist.index(minTempDist)])
102
103   resDistMin = min(distMin)
104
105   ladoPonto = indices[distMin.index(resDistMin)]
106
107   return resDistMin, indices.index(ladoPonto), ladoPonto
108
```

Listing 3: Função distanciasPO

- **Distância Cosseno** - tipo de métrica que avalia o grau de abertura entre dois vetores que possuem a mesma origem. No caso em questão será medida a abertura entre o vetor distância até o obstáculo e o vetor distância até o *target*. É calculada pela biblioteca `scipy.spatial.distance` chamando a função `cosine`:

```
1 def calculaDistCosseno(point, obst, target):
2   u = np.array(obst) - np.array(point)
3   v = np.array(target) - np.array(point)
```

```
4
5   return ssd.cosine(u, v)
6
```

Listing 4: Função calculaDistCosseno

- **Seleção do ponto de menor distância do obstáculo** - seleciona a coordenada do ponto de menor distância do obstáculo mais próximo:

```
1  def calculaPontoObst(point):
2    #Seleciona o obstaculo
3    seletorObst = distanciasPO(point)[1]
4    obstaculo = ""
5
6    if seletorObst == 0:
7      obstaculo = OBS1
8    elif seletorObst == 1:
9      obstaculo = OBS2
10   elif seletorObst == 2:
11     obstaculo = OBS3
12   elif seletorObst == 3:
13     obstaculo = OBS4
14   elif seletorObst == 4:
15   obstaculo = OBS5
16     elif seletorObst == 5:
17   obstaculo = OBS6
18
19   #Seleciona o lado
20   ladoObst = distanciasPO(point)[2][0]
21
22   #Seleciona o ponto no lado
23   pontoObst = distanciasPO(point)[2][1]
24
25   return obstaculo[ladoObst-1][pontoObst]
26
```

Listing 5: Função calculaPontoObst

2.3. Desenvolvimento do Algoritmo Genético

Elaboramos nosso *GA* baseado no modelo dado em sala de aula mas seguindo o seguinte paradigma biológico: **uma população é formada por indivíduos, que são formados por cromossomos, que são compostos de genes**. Dessa forma, para efeitos de algoritmo, definimos os seguintes termos:

- **Gene:** uma coordenada individual **x** ou **y**;

- **Cromossomo:** um par ordenado formado por duas coordenadas **(x,y)**;
- **Indivíduo:** uma lista de **k** cromossomos;
- **População:** uma lista de **n** indivíduos.

Dessa forma, para nós, uma solução se configura em um caminho entre a posição inicial do robô(**start**) e o destino(**target**), o que é, para efeitos de *GA*, um indivíduo dentro da população que possua a maior fitness ao final de **n** gerações.

Definimos um cromossomo conforme o código abaixo:

```
1  # -*- coding: UTF-8 -*-
2  import numpy as np
3  import scipy.spatial.distance as ssd
4  import obstaculos
5  from intervals import FloatInterval as fInterval
6
7  def generateChromosome():
8    x = np.random.uniform(0, 5, 1)
9    y = np.random.uniform(0, 5, 1)
10   return np.array((round(x[0], 1), round(y[0], 1)))
11
```

Listing 6: Função generateChromosome

Nesta função, um gene é definido como um número aleatório de distribuição uniforme de probabilidade entre 0 e 5, correspondendo às dimensões do mapa.

A seguir geramos um indivíduo conforme o código abaixo:

```
1  def generateIndividual(tamInd):
2    individuo = []
3
4    for i in range(0, tamInd):
5      individuo.append(generateChromosome())
6
7    return individuo
8
```

Listing 7: Função generateIndividual

Passamos como parâmetro o tamanho desejado para o indivíduo, ou seja, quantos pontos(cromossomos) entre *start* e *target* formarão o caminho(**path**).

Para a formação da população, desenvolvemos o seguinte código:

```
def generatePopulation(nIndividuals, tamInd):
  pop = []

  for i in range(0, nIndividuals):
    pop.append(generateIndividual(tamInd))

  return pop

```

Listing 8: Função generatePopulation

Como parâmetros para esta função temos o tamanho desejado para a população e o tamanho do indivíduo da população.

A função considerada mais importante para a execução do *GA* é a **fitness**, que irá avaliar o grau de adaptação do indivíduo como solução do problema. Nossa função, então, avalia a fitness de cada indivíduo considerando que ela é **diretamente proporcional à soma das distâncias aos obstáculos** - cada distância ao obstáculo é ponderada pelo grau de abertura do ângulo entre a distância ao obstáculo e a distância ao *target*(avalia se há perigo de colisão caso prossiga direto) - e **inversamente proporcional ao comprimento do indivíduo(soma das distâncias entre os pontos)**. Segue o código da fitness:

```
def fitness(individuo, start, target):
lenPath = obstaculos.distanciaPontos(start, individuo[0])
lenDistObst = 0

for cromossomo in range(0, len(individuo) - 1):
  #Calcula a distancia do segmento ate o proximo ponto e
    adiciona a lenPath
  lenPath += obstaculos.distanciaPontos(individuo[cromossomo],
    individuo[cromossomo+1])

  #Calcula a distancia ate o obstaculo mais proximo do
    cromossomo e adiciona a lenDistObst
  distCromObs = obstaculos.distanciasPO(individuo[cromossomo])
    [0]

  #Calcula o cosseno entre a distancia ao obstaculo e a
    distancia ao target
  pontoObst = obstaculos.calculaPontoObst(individuo[cromossomo])
  distCosseno = obstaculos.calculaDistCosseno(individuo[
    cromossomo], pontoObst, target)

  lenDistObst += distCromObs * distCosseno

lenPath += lenPath + obstaculos.distanciaPontos(individuo[len(
```

```
    individuo) - 1], target)

return lenDistObst/lenPath

```

Listing 9: Função fitness

Como método de seleção optamos pelo uso a **Roleta**, onde os indivíduos com maiores fitness tem maior probabilidade de serem selecionados para reprodução. É calculada a fitness total da população e as fitness relativas de cada indivíduo em relação à fitness total. Como metódo de escolha geramos um numero aleatório entre 0 e a maior fitness relativa, estabelecendo previamente os intervalos dentro da fitness total e escolhemos o indivíduo cujo intervalo contemple o número sorteado. Eis o código:

```
def roleta(populacao, start, target):
  #Calcula um vetor de fitness dos individuos da populacao
  fitPop = []
  porcFitness = []
  sumFitness = 0


  for individuo in populacao:
    fitIndividuo = fitness(individuo, start, target)
    fitPop.append(fitIndividuo)
    sumFitness += fitIndividuo

  #Calcula a porcentagem de cada fitness individual em relacao a
     fitness total
  for fit in fitPop:
    porcFitness.append(round(fit/sumFitness,2))

  # Calcula a posicao da agulha da roleta - numero entre 0 e 1
    randomico
  agulha = np.random.uniform(0, max(porcFitness), 1)/sumFitness

  #Cria os intervalos da roleta e faz o giro
  resultado = 0
  a = 0
  for porc in porcFitness:
    b = a + porc
    if float(agulha) in fInterval.closed(a, b):
      break
    else:
      resultado += 1
    a = b

  return populacao[resultado]
```

```
32
```

Listing 10: Função roleta

O cruzamento ocorre através do método **Ponto Único**, onde o conjunto de cromossomos do indivíduo é dividido em duas partes no ponto definido e feito o intercâmbio com a outra parte proveniente de outro indivíduo selecionado e junção no mesmo ponto das partes trocadas.

```
1  def crossover(indivA, indivB, ponto):
2
3    crossAB = []      #Primeiro descendente
4    crossBA = []      #Segundo descendente
5
6    #Faz os cruzamentos
7    for i in range(0, int(ponto)):
8      crossAB.append(indivA[i])
9      crossBA.append(indivB[i])
10
11   for j in range(int(ponto), len(indivA)):
12     crossAB.append(indivB[j])
13     crossBA.append(indivA[j])
14
15   return crossAB, crossBA
16
```

Listing 11: Função crossover

Implementamos a **mutação** selecionando um cromossomo do indivíduo aleatoriamente e trocando as coordenadas de posição dentro do mesmo:

```
1  def mutacao(individuo):
2    mut = np.random.randint(0, len(individuo))
3
4    temp = individuo[mut][0]
5    individuo[mut][0] = individuo[mut][1]
6    individuo[mut][1] = temp
7
8    mutante = individuo
9
10   return mutante
11
```

Listing 12: Função mutação

Por fim, a função **run** executa o *GA* com os parâmetros devidamente passados para a referida função(ponto de início, destino, tamanho do indivíduo, número de gerações e probabilidade de mutação):

```
def run(start, target, tamPop, tamIndividuo, nGeracoes,
    probMutacao):
  fitInicial = []
  proximaPopulacao = []
  fitnessIndividuo = 0
  geracao = 0

  #Gere uma populacao de individuos aleatorios
  print "Gerando populacao com " + str(tamPop) + " individuos...
    "
  popIncial = generatePopulation(tamPop, tamIndividuo)

  #Verifique a fitness de cada individuo da populacao inicial
  print ""
  print "Calculando a fitness inicial da populacao..."
  for individuoInicial in popIncial:
    fitnessIndividuo = fitness(individuoInicial, start, target)
    fitInicial.append(fitnessIndividuo)

  #Crie a proxima populacao
  #Garante na proxima populacao a maior fitness
  print ""
  print "Criando a proxima populacao..."

  while geracao < nGeracoes:
    proximaPopulacao = []
    print str(geracao + 1) + "a geracao..."
    proximaPopulacao.append(popIncial[fitInicial.index(max(
    fitInicial))])
    print proximaPopulacao, fitInicial.index(max(fitInicial))

    for i in range(0, int(np.floor(len(popIncial) / 2))):
      # Seleciona um par de individuos da populacao para ser os
    pais
      paiA = roleta(populacao=popIncial, start=start, target=
    target)
      paiB = roleta(populacao=popIncial, start=start, target=
    target)

      # Gera os filhos e adiciona para a proxima populacao,
    cruzando-os pelo meio
      filhos = crossover(paiA, paiB, np.floor(len(paiA) / 2))
      proximaPopulacao.append(filhos[0])
      proximaPopulacao.append(filhos[1])

    popIncial = proximaPopulacao

    #Determina se vai haver alguma mutacao na populacao
    probMut = np.random.randint(1,10,1)/100
```

```
43      if probMut == probMutacao:
44        mutante = np.random.randint(0, len(popIncial),1)
45        mutacao(popIncial[mutante])
46
47      geracao += 1
48
```

Listing 13: Função run

Como critério de parada utilizamos um determinado número de gerações, o qual é passado para a função.

3. Discussão

Tivemos problemas em testar efetivamente o algoritmo no VREP pois inicialmente não conseguíamos extrair as informações de posição dos obstáculos e de distância do robô ao obstáculo do ambiente de simulação para passar para o GA, motivo pelo qual tivemos que desenvolver a biblioteca de funções `obstaculos.py`, para abstrair as informações do mapa. Chegamos inclusive a contactar, via email, o professor **Eric Rohmer**, da Universidade de Campinas(`http://www.dca.fee.unicamp.br/~eric/`), colaborador da Coppelia Robotics e criador do Pioneer_p3dx, o qual gentilmente nos respondeu e sugeriu algumas medidas, as quais tentamos implementar mas, mesmo assim não obtivemos sucesso.

4. Conclusão

Concluímos que *GA*'s são uma estratégia que podem ser implementadas com relativo sucesso em problemas de *path planning* como o proposto, ainda que nem sempre traga consigo a melhor resposta ao problema, e o trabalho em questão serviu para aprimorarmos o conteúdo ministrado em sala de aula, mesmo que não tenhamos tido a oportunidade testá-lo no robô.