Biologia Computacional - Lista VI

Nome: Rodrigo Okido - 252745
Prof.: Márcio Dorn

Sumário

Algoritmos Implementados

- Simulated Annealing para busca em trajetória
  - Baseado em temperatura
  - “Resfriamento”
  - Permite obter soluções que possam sair do ótimo local
- Genéticos
  - Geração de população inicial
  - Crossover
  - Mutação

Exercicio 1/2 - Item a

```
#List of locations passed
locations = []
#Limit of iterations to reach the ideal solution
run_limit = 50000
#Euler constant user for Item A of List VI
euler_a =  2.718


#Function for item A
def check_function_himmelblau(x, y):
    return math.pow((math.pow(x,2)+y-11),2) + math.pow((x+math.pow(y,2)-7),2)


###################### SIMULATED ANNEALING ######################
# Algoritmo de Trajetoria (Item A do exercicio 1 da lista VI)


#Generate a random neighbor. This is a function to be used in Simulated Annealing.
def generate_neighbor():
    pos_x = random.uniform(-10.0,10.0)
    pos_y = random.uniform(-10.0,10.0)
    return pos_x,pos_y
```

Exercicio 1 - Item a

```
#Function to determine the cost to move for the next position. Function to be
#used in Simulated Annealing.
def costA(x, y):
    return check_function_himmelblau(x,y)


#Funcion to verify the probability of the new solution to be accepted or not.
#Function used in Simulated Aneealing.
def acceptance_probability (old_cost, new_cost, temp):
    if new_cost < old_cost:
        return 1.0
    else:
        calculate_exp = (new_cost - old_cost) / temp
        if calculate_exp < 0:
            new_c = round(math.pow(euler_a,calculate_exp),2)
            if new_c < 0.01:
                return 0
            else:
                return new_c
        else:
            return 0

```

Exercicio 1/2 - Item a

```
#Simulated Annealing Function.
def simulated_annealing():
    first_x = random.uniform(-10.0,10.0)
    first_y = random.uniform(-10.0,10.0)
    old_sol = costA(first_x, first_y)


    #Setting parameters values (Temp, Temp_min and Alpha)
    temp = 1.0
    temp_min = 0.000001
    alpha = 0.95
    run_times = 0
    while temp > temp_min or run_times < run_limit:
        i = 1
        while i <= 100:
            new_x, new_y = generate_neighbor()
            new_sol = costA(new_x, new_y)
            ap = acceptance_probability(old_sol, new_sol, temp)
            if ap > 0.6:
                sol = round(new_sol,2)
                locations.append(sol)
                old_sol = new_sol
            i += 1
            run_times += 1
        temp = temp*alpha

    #print run_times
    return locations
```

Exercicio 1/2 - Item b

```
79  ###################### GENETIC ALGORITHMS ######################
80  # Algoritmo baseado em populacaso (Item B do exercicio 2 da lista VI)
81
82  population_A = []
83  population_B = []
84  population_X = []
85
86  class Pair:
87    def __init__(self, x, y):
88      self.x = x
89      self.y = y
90
91  def generate_population():
92      for i in range(0,100):
93          gen_x = random.uniform(-10.0,10.0)
94          gen_y = random.uniform(-10.0,10.0)
95          generated = Pair(gen_x,gen_y)
96          population_A.append(generated)
97
98          #Checks for elites
99          check_pair = calc_pairs(generated.x , generated.y)
100         if(check_pair > -0.2 and check_pair < 0.2):
101             population_X.append(generated)
102
103     for j in range(0,100):
104         gen_x = random.uniform(-10.0,10.0)
105         gen_y = random.uniform(-10.0,10.0)
106         generated = Pair(gen_x,gen_y)
107         population_B.append(generated)
108
```

Exercicio 1/2 - Item b

```
def cross_population():
    counter = 0
    run_times = 0
    pop_1 = population_A
    pop_2 = population_B
    pop_generated = []
    while counter != 100 and run_times < run_limit:
        pop_generated = []
        pop_control = 0
        for i in range(len(pop_1)):
            new_pair = Pair(pop_1[i].x, pop_2[i].y)
            val = calc_pairs(new_pair.x, new_pair.y)
            if val > -0.2 and val < 0.2:
                population_X.append(new_pair)
                pop_generated.append(new_pair)
            else:
                mutated = mutation(new_pair)
                pop_generated.append(new_pair)
            if counter == 100:
                break

        if pop_control == 0:
            pop_2 = pop_generated
            pop_control = 1
        else:
            pop_1 = pop_generated
            pop_control = 0
        run_times += 1


def mutation(pair):
    mut_pair = pair
    mut_prob = random.uniform(0.0,1.0)
    if mut_prob > 0.75:
        mut_pair.x = random.uniform(-10.0,10.0)
        mut_pair.y = random.uniform(-10.0,10.0)
        return mut_pair
    return mut_pair
```

Exercicio 1/2 - Item b

```
148 def calc_pairs(x,y):
149     return check_function_ackley(x,y)
150
151
152 #Main function
153 def main():
154     # simulated_annealing()
155     # print locations
156
157     generate_population()
158     cross_population()
159     for j in range(len(population_X)):
160         print "( ",population_X[j].x, ",", population_X[j].y ," )"
161
162 if __name__ == "__main__":
163     main()
164
```

Média e Desvio Padrão

Ambos calculados nas suas formas tradicionais (Pega todos elementos, e divide pela quantidade total).

```python
def calc_media(lista):
    size = len(lista)
    media_x = 0
    media_y = 0
    for i in range(len(lista)):
        media_x += lista[i].x
        media_y += lista[i].y
    return media_x/size, media_y/size

def calc_desvio_padrao(lista):
    media_x,media_y = calc_media(lista)
    dv_x = 0
    dv_y = 0
    for i in range(len(lista)):
        dv_x += math.pow(lista[i].x - media_x,2)
        dv_y += math.pow(lista[i].y - media_y,2)
    return math.sqrt(dv_x/len(lista)), math.sqrt(dv_y/len(lista))
```

Resultados

Item A - 1

- Output (Possível Resultado):
    - [5836.55, 146.14, 67.94, 59.41, 27.56, 16.43, 2.26, 0.39, 0.19, 0.13, 0.02]

Item A - 2

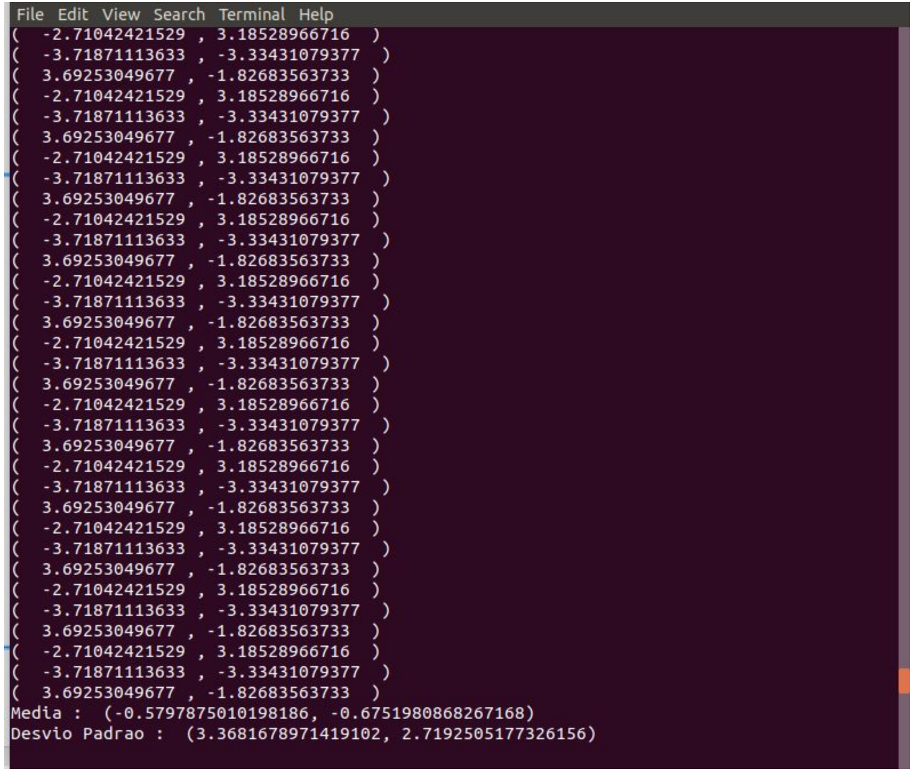
- Output (Possível Resultado):
    - [-88.86, -8.25, -7.43, -4.69, -3.04, -2.14, -2.23, -2.14, -2.35, -1.78, -1.65, -1.46, -1.47, -1.45]

Ambos usam o limite de 50.000 iterações. No primeiro caso, o resultado vem mais corretamente na maioria dos casos, enquanto no segundo oscila dependendo do grau de probabilidade de aceitação.

Resultados

Item B - 1

Média (X,Y) e Desvio Padrão (X,Y)



Resultados

Item B - 2

Algoritmo não consegue convergir para resultados bons.

Em ambos os casos os resultados acabaram não sendo exatamente o melhor. Entretanto, apenas o item A foi possível extrair algum resultado para realizar uma análise da mesma. No item B, os descendentes nunca ficam bons o suficiente para se aproximar do resultado de 0 da função. Com isto acaba estourando o limite de iteração (50.000), retornando -1.

Obrigado!