Relatório do Laboratório 4: Otimização com Métodos Baseados em População

Isabelle Ferreira de Oliveira
CT-213 - Engenharia da Computação 2020
Instituto Tecnológico de Aeronáutica (ITA)
São José dos Campos, Brasil
isabelle.ferreira3000@gmail.com

***Resumo*—Esse relatório documenta a implementação do algoritmo de otimização baseado em população: *Particle Swarm Optimization* (PSO). Esse método foi testado tanto na otimização de uma função simples, para verificar o correto funcionamento do PSO, quanto na otimização dos parâmetros do controlador de um robô seguidor de linha.**

***Index Terms*—Algoritmos de Otimização, população, *Particle Swarm Optimization* (PSO), robô seguidor de linha**

I. Introdução

Otimização consiste em encontrar o mínimo (ou máximo) de uma função, ou seja, encontrar o conjunto de parâmetros que levem essa função ao seu mínimo (ou máximo). Também pode ser visto como encontrar a melhor solução dentre todas as soluções viáveis. Nesses problemas de otimização também é possível haver restrições acerca dos parâmetros que serão analisados.

Dentre os mais diversos tipos de algoritmos de otimização, existem os métodos baseado em população. Esses métodos mantêm uma população de possíveis soluções, a fim de tentar encontrar diferentes mínimos locais da função a ser analisada, melhorando assim as chances de se obter uma solução promissora. Um exemplo famoso de algoritmo de otimização baseado em população é o *Particle Swarm Optimization* (PSO).

O pseudo-código desse algoritmo para maximização pode ser visto na subseção a seguir. Em seguida, será apresentado como esse algoritmo foi implementado no contexto do laboratório.

A. Particle Swarm Optimization (PSO)

Será considerado uma classe *ParticleSwarmOptimization*, representando o algoritmo PSO. Além disso, o pseudocódigo do uso do algoritmo a partir dessa classe é o mostrado a seguir. Nesse pseudocódigo, *pso* é o objeto da classe *ParticleSwarmOptimization*.

```
for i in range(num_evaluations):
  position = pso.get_position_to_evaluate()
  value = quality_function(position)
  pso.notify_evaluation(value)
```

No pseudocódigo acima, *quality_function()* é a função a ser maximizada. Assim, iterativamente por tantas vezes até se atingir uma convergência satisfatória ao(à) programador(a), o algoritmo calculará o valor da função para cada partícula na população de candidatas à solução. Para cada partícula, então, após esse cálculo, a função *notify_evaluation()* compara o resultado obtido aos obtidos anteriormente a fim de encontrar o valor maximizado. Uma ideia de implementação em pseudocódigo da função *notify_evaluation()* foi apresentada a seguir.

```
def notify_evaluation(value):
  current_particle =
      get_current_particle_to_evaluate()

  if value > current_particle.my_best_value:
    current_particle.my_best_value = value
    current_particle.my_best_position =
        current_particle.position

  if value > global_best_value:
    global_best_value = value
    global_best_position =
        current_particle.position

  num_particle_evaluated =
      num_particle_evaluated + 1
  if num_particle_evaluated == num_particles:
    num_particle_evaluated = 0
    advance_generation()
```

A função *advance_generation()* é a responsável por atualizar os valores de cada partícula a cada geração de população. As posições de cada particula são acrescidas de uma velocidade para gerar as posições a serem analisadas na geração seguinte, e o intuito dessa velocidade é conduzir as partículas a posições cada vez mais próximas da solução otimizada.

Essa velocidade foi calculada a partir da equação fornecida a seguir, na qual ω é relativo a inércia dessa mudança, φ_p e φ_g são, respectivamente, os parâmetros cognitivo e social, e b e b_g são, respectivamente, as melhores posições a nível dessa partícula em específica e a nível global (entre todas as partículas). Além disso, existem os parâmetros r_p e r_g, que conferem aleatoriedade ao sistema.

$$v = \omega \cdot v + \varphi_p \cdot r_p \cdot (b - x) + \varphi_g \cdot r_g \cdot (b_g - x)$$

Abaixo, por fim, está apresentado o pseudocódigo para a ideia acima.

```
def advance_generation():
  for particle in particles:
    r_p = random.uniform(0, 1)
```

```
r_g = random.uniform(0, 1)

particle.velocity = inertia_weight *
    particle.velocity +
    cognitive_parameter * r_p *
    (particle.my_best_position -
    particle.position) + social_parameter
    * r_g * (global_best_position -
    particle.position)

for i in range(quantity_of_dimensions):
  delta = upper_bound[i] - lower_bound[i]
  particle.velocity[i] =
      min(max(particle.velocity[i],
      -delta), delta)

particle.position = particle.position +
    particle.velocity

for i in range(quantity_of_dimensions):
  particle.position[i] =
      min(max(particle.position[i],
      lower_bound[i]), upper_bound[i])
```

II. Implementação dos algoritmos

Na parte relativa a implementação dos algoritmos de otimização, era necessário preencher os códigos das funções *gradient_descent()*, *hill_climbing()* e *simulated_annealing()* do código base fornecido [1]. Além disso, era necessário completar também os códigos das funções *neighbors()* (para o método Hill Climbing), *random_neighbor()* e *schedule()* (para o método Simulated Annealing).

A análise de vários pontos dos algoritmos descritos acima terão uma breve descrição em alto nível da sua implementação a seguir.

Primeiramente, foi criada uma função *check_stopping_condition()*, que, a partir do valor da função de custo naquele determinado teste, de um limite mínimo aceitável para essa função de custo, além dos números de iterações máximos aceitáveis e qual o atual número de iteração, decidia se era situação de parar o algoritmo ou não.

As funções *J* apresentadas nos pseudo códigos acima se referiam a própria função de custo para cada um dos métodos e a função *dJ* ao gradiente dessa função (esse já fornecido pelo código base).

Já a função *neighbors()* retornava um *array* de parâmetros vizinhos ao parâmetro analisado nessa determinada iteração, e esses vizinhos eram calculados conforme descrito no roteiro do laboratório [1], utilizando as projeções no eixo X e Y (calculadas em Python a partir de *numpy.cos()* e *numpy.sin()*) para encontrar as coordenadas de cada vizinho.

A função *random_neighbor()* foi implementada de forma análoga. A grande diferença foi, ao invés de retornar um *array* de vizinhos, retornava apenas um, e o ângulo para as projeções foi obtido aleatoriamente de forma uniforme a partir da função de Python *random.uniform(-numpy.pi, numpy.pi)*.

A função *schedule()* seguiu a ideia fornecida pelo roteiro [1]. Assim, essa função retornava em Python *temperature0/(1 + beta * (i ** 2))*.

Por fim, as funções *gradient_descent()*, *hill_climbing()* e *simulated_annealing()* foram implementadas conforme apresentado nos pseudo códigos da seção Introdução, com alguns detalhes como armazenar toda a trajetória dos métodos adicionando os parâmetros *theta* testados a cada iteração na lista *history*. Além disso, para o caso do Hill Climbing, foram adicionadas condições específicas relativas ao caso no qual a variável *best* ainda fosse *None*.

III. Resultados e Conclusões

Os resultados das trajetórias de otimização obtidos após a execução das implementações dos algoritmos descritos acima foram apresentados nas Figuras 1, 2 e 3 para Descida do Gradiente, Hill Climbing e Simulated Annealing, respectivamente, e a sobreposição dessas trajetórias foi apresentada na Figura 4 para melhor comparação visual.

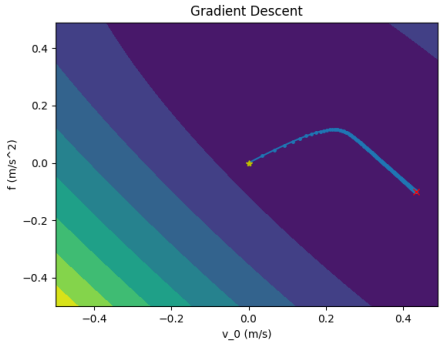

Figura 1. Trajetória de otimização usando Descida do Gradiente.

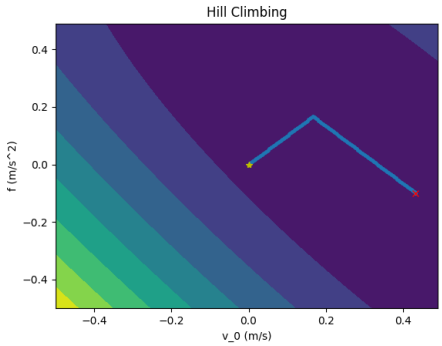

Figura 2. Trajetória de otimização usando Hill Climbing.

Aplicando os valores encontrados para os parâmetros v_0 e f, o gráfico de velocidade da bola por tempo foi apresentado na Figura 5. Esses valores numéricos de v_0 e f podem ser vistos na Tabela I para cada método estudado nesse laboratório, além do resultado fornecido inicialmente para o método de Mínimos Quadrados.

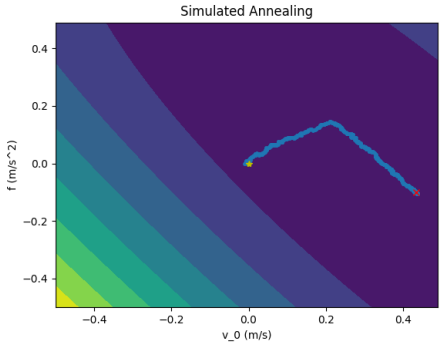

Figura 3. Trajetória de otimização usando Simulated Annealing.

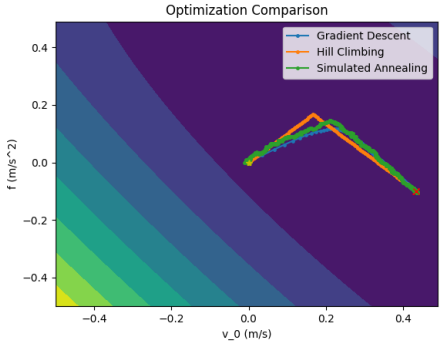

Figura 4. Comparação de trajetórias de otimização usando Descida do Gradiente, Hill Climbing e Simulated Annealing.

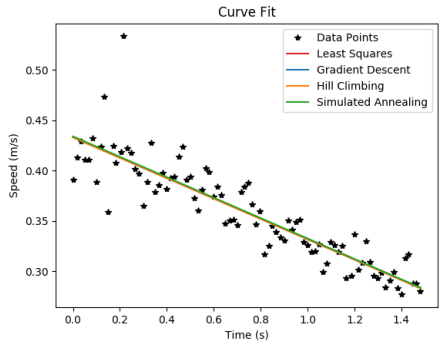

Figura 5. Comparação das regressões lineares das otimizações usando Descida do Gradiente, Hill Climbing e Simulated Annealing.

Tabela I
COMPARAÇÃO DAS SOLUÇÕES ENCONTRADAS PARA OS PARÂMETROS FÍSICOS DA BOLA PARA OS ALGORITMOS IMPLEMENTADOS NO LABORATÓRIO.

Algoritmo	**Solução**	
	v_0	f
Mínimos Quadrados	0.43337277	-0.10102096
Descida do Gradiente	0.4333707	-0.10101849
Hill Climbing	0.43274935	-0.10099495
Simulated Annealing	0.43397656	-0.10134529

Tendo em vista o que foi apresentado, pode-se notar, por fim, que esses algoritmos realmente se demonstraram eficazes em encontrar parâmetros otimizados para uma determinada função de custo e um ponto inicial de partida.

REFERÊNCIAS

[1] M. Maximo, "Roteiro: Laboratório 3 - Otimização com Métodos de Busca Local". Instituto Tecnológico de Aeronáutica, Departamento de Computação. CT-213, 2019.