UNIVERSIDADE
DE TRÁS-OS-MONTES
E ALTO DOURO

2023/2024

Algoritmos de Pesquisa

Engenharia Informática

Inteligência Artificial

Corpo Docente:
Paulo Moura Oliveira
Eduardo Solteiro Pires

PL3
al70759 Cláudia Oliveira da Silva
al71134 Pedro Miguel Monteiro Barroso

Resumo

Este trabalho explora três algoritmos de otimização - Hill Climbing, Simulated Annealing e Algoritmo Genético - aplicados à função objetivo específica. Inicialmente, é feita uma pesquisa de informações e análise dos algoritmos. O foco está em implementar e testar cada algoritmo para encontrar o máximo global da função. O trabalho prático inclui a programação destes algoritmos em MATLAB, respeitando parâmetros como número de gerações, tamanho do cromossoma, probabilidade de cruzamento e mutação. Por fim, é apresentando os resultados gráficos e análises dos testes realizados.

Índice

1. Introdução

1.1 Objetivos do Projeto:

- **Compreender e implementar os algoritmos de Hill Climbing, Simulated Annealing e Algoritmo Genético.**
- **Aplicar esses algoritmos na otimização de uma função objetivo específica.**
- **Analisar e comparar o desempenho de cada algoritmo em termos de eficácia e eficiência.**
- **Desenvolver habilidades práticas em programação e resolução de problemas em MATLAB.**

2. Algoritmos de Pesquisa

Os algoritmos de pesquisa representam métodos computacionais fundamentais para resolver problemas que envolvem a busca ou otimização de soluções. Estas técnicas variam desde abordagens simples, como a busca linear, até métodos mais complexos, como algoritmos genéticos e redes neurais. No contexto da otimização, os algoritmos de pesquisa exploram sistematicamente o espaço de soluções para encontrar a melhor solução possível. Eles são essenciais em várias áreas, incluindo inteligência artificial, otimização matemática e processamento de dados, e são cruciais para a resolução eficiente de problemas complexos.

2.1 Hill-Climbing

O Hill Climbing é um algoritmo de busca local, fundamental no campo da otimização computacional. Este método é frequentemente comparado ao processo de escalar uma colina, onde o objetivo é alcançar o ponto mais alto. No contexto computacional, isto traduz-se em encontrar a solução ótima ou a mais próxima possível dela para um dado problema

Funcionamento do Algoritmo Hill-Climbing

O algoritmo inicia com uma solução aleatória, que pode ser um ponto num espaço de busca multidimensional. A partir desse ponto, o Hill Climbing avalia as soluções vizinhas e se move na direção daquela que apresenta a maior melhoria (ou, dependendo do problema, a

menor degradação) em relação à função objetivo. Este processo é iterativo e continua até que não sejam encontradas melhorias nas proximidades, indicando que o algoritmo alcançou um máximo local.

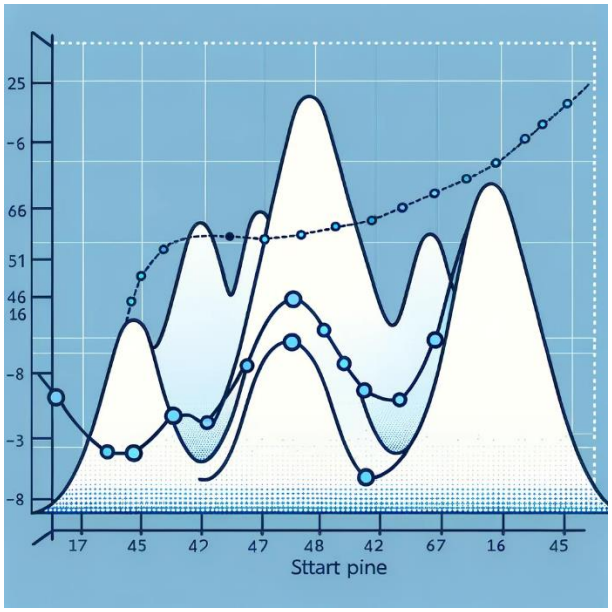


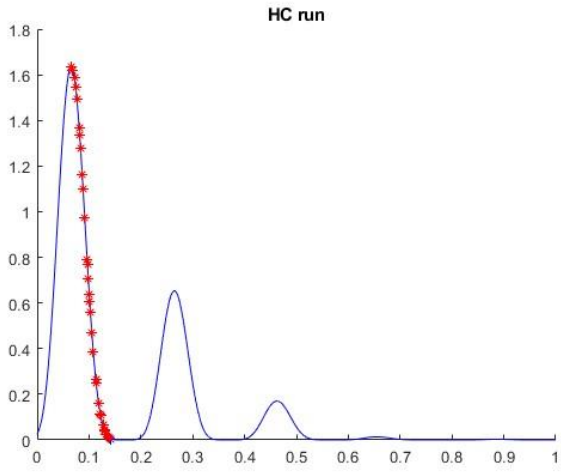
Figura 1-A imagem ilustra o conceito do algoritmo Hill Climbing.

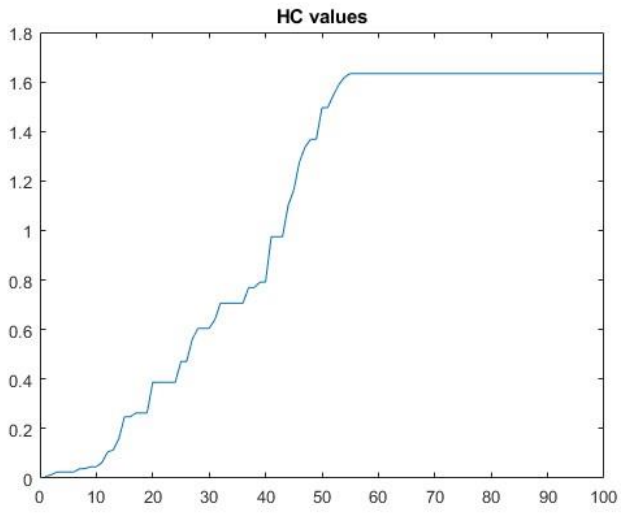
Limitações e Considerações

Uma das principais limitações do Hill Climbing é a sua tendência a ficar preso em máximos locais, especialmente em paisagens com muitos picos e vales. Não há garantia de encontrar o máximo global, a menos que o ponto de partida esteja próximo dele ou a paisagem seja suave sem muitos máximos locais.

Apesar dessas limitações, o Hill Climbing é apreciado pela simplicidade e eficácia em muitos problemas práticos, especialmente quando combinado com outras estratégias para evitar armadilhas de máximos locais.

Analise da solução (ANEXO A)





O primeiro gráfico mostra a trajetória do algoritmo sobre a paisagem da função objetivo. As estrelas vermelhas representam a posição do algoritmo em cada iteração, indicando claramente a abordagem incremental típica do Hill Climbing. Observa-se que o algoritmo começou num ponto baixo, escalou a encosta mais próxima e aparentemente atingiu um máximo local. Este comportamento é característico do Hill Climbing, onde o algoritmo "sobe" até encontrar um pico, mesmo que não seja o máximo global.

No segundo gráfico, temos a evolução do valor da função objetivo ao longo das iterações do algoritmo. Este gráfico de linha mostra que houve uma tendência consistente de melhoria ao longo do tempo, com alguns patamares onde o algoritmo não conseguiu encontrar imediatamente uma direção de melhoria, o que é esperado devido à natureza exploratória do algoritmo em busca local.

2.2 Simulated Annealing

O Simulated Annealing é um algoritmo probabilístico inspirado no processo de recozimento da metalurgia. Este método procura soluções ótimas em problemas de otimização global, simulando o processo físico de aquecimento e arrefecimento gradual de um material para alcançar um estado de energia mínima.

Funcionamento do Simulated Annealing

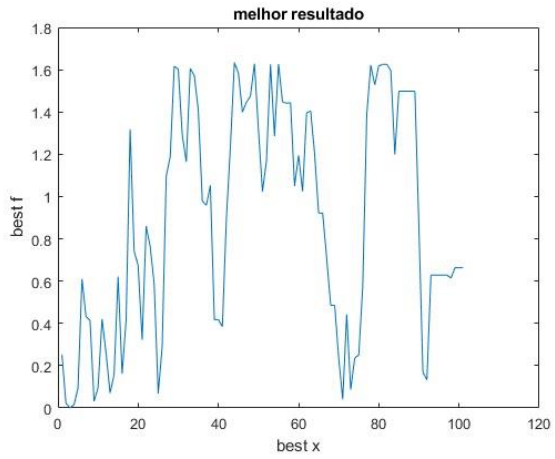
O algoritmo inicia com uma temperatura inicial elevada que vai sendo gradualmente arrefecida. Em cada passo, gera-se uma nova solução através de uma pequena alteração da solução atual. Se esta nova solução for melhor, é aceite. Caso seja pior, ainda pode ser aceite

com uma probabilidade que depende da diferença de qualidade entre as soluções e da temperatura corrente. A aceitação de soluções piores ajuda a evitar ficar preso em mínimos locais, tornando o Simulated Annealing eficaz para encontrar mínimos globais em paisagens complexas.

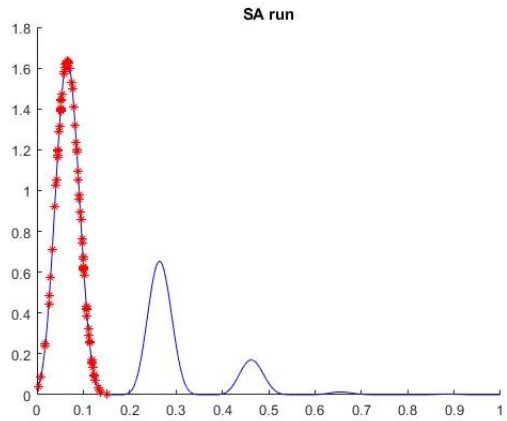
Aplicações e Limitações

Utilizado numa variedade de problemas, desde a otimização de rotas até ao design de circuitos, o Simulated Annealing revela-se particularmente útil em situações com muitos mínimos locais. No entanto, a escolha de parâmetros como a taxa de arrefecimento e a temperatura inicial é crucial para o seu desempenho e pode necessitar de ajustes específicos para cada problema.

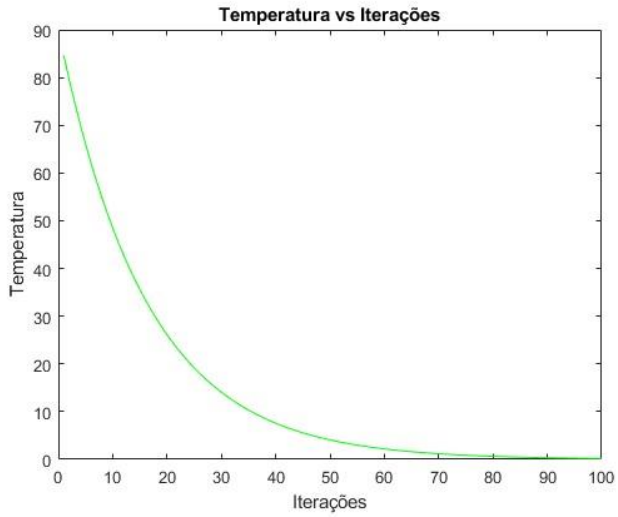
Analise da solução (ANEXO B)



O primeiro gráfico, "melhor resultado", mostra como a qualidade da melhor solução que o Simulated Annealing encontrou vai mudando com cada tentativa. O gráfico mostra subidas e descidas, o que indica que o algoritmo não se contenta apenas com a primeira boa solução que encontra, mas continua a procurar por outras, possivelmente melhores. Esta característica é importante porque ajuda o algoritmo a não se fixar apenas em soluções "boas o suficiente" e a procurar sempre melhorar.



O segundo gráfico, "SA run", mostra o caminho que o algoritmo percorreu na tentativa de encontrar a melhor solução. As marcas vermelhas mostram onde o algoritmo parou para verificar se tinha encontrado uma boa solução. Estas marcas estão frequentemente perto dos picos do gráfico, o que sugere que o algoritmo está a fazer um bom trabalho na procura das melhores soluções.



O terceiro gráfico, "Temperatura vs Iterações", ilustra como a "temperatura" do algoritmo vai descendo com o tempo. No início, o algoritmo tem mais "energia" para explorar, mas à medida que o tempo passa, ele começa a acalmar e a concentrar-se nas soluções que parecem ser as melhores. A linha que desce gradualmente mostra que esta redução de "energia" é feita de forma controlada.

2.3 Genetic Algorithm

Os Algoritmos Genéticos são métodos de busca e otimização inspirados nos princípios da genética e seleção natural. São particularmente eficazes na resolução de problemas complexos onde outras abordagens podem falhar.

Funcionamento do Algoritmo Genético

No Algoritmo Genético, uma população de soluções candidatas (cromossomas) evolui ao longo das gerações. Cada cromossoma é avaliado por uma função de aptidão. As operações de seleção, cruzamento (reprodução) e mutação são aplicadas para criar gerações. A seleção prioriza indivíduos mais aptos, enquanto o cruzamento e a mutação introduzem diversidade genética na população, permitindo a exploração de novas áreas do espaço de pesquisa.

Aplicações e Eficiência

Os Algoritmos Genéticos são amplamente utilizados numa diversidade de campos, incluindo na otimização de problemas complexos, design de sistemas, inteligência artificial e aprendizagem automática. A sua eficácia advém da capacidade de explorar e aproveitar simultaneamente o espaço de pesquisa, encontrando soluções ótimas em paisagens de pesquisa complexas e variadas.

Analise da solução (ANEXO C)

3. Conclusão

Chegando ao fim deste projeto, exploramos três maneiras diferentes de resolver problemas difíceis de otimização. O Hill Climbing é fácil de entender e usar, mas às vezes não consegue encontrar a melhor solução possível. O Simulated Annealing foi um pouco mais complexo, mas conseguiu evitar ficar preso em soluções que não eram as melhores.

Com o Algoritmo Genético, tivemos alguns problemas, especialmente para entender que gráficos desenvolver. Isso mostrou que este algoritmo é mais complicado, pois tenta imitar como a natureza escolhe as melhores características ao longo do tempo. Mesmo com essas dificuldades, aprendemos bastante com as experiências que tivemos.

ANEXO A

```
clear("all");
f = @(x) 4 * sin(5 * pi * x + 0.5) .^ 6 .* exp(log2((x - 0.8) .^ 2));
x0 = rand();
max_iters = 100;

%Hill Climb
[x_best, f_best, best_hc] = HillClimb(f, x0, 0, 1, max_iters);
hc_data = [x_best, f_best, best_hc];
figure
plot(best_hc);
title("HC values")

function [x_best, f_best, best_hc] = HillClimb(f, x0,lower,upper, max_iters)

figure
    hold on
    title("HC run")

    x=0:10^-5:1;
    fx = f(x);
    plot(x,fx,'b');
    x_best = x0;
    f_best = f(x_best);
    best_hc = [];
    plot(x_best,f_best,'*b');
    for i = 1:max_iters
        r = (rand() - 0.5)*10^-2;
        x_new = x_best + r;
        while x_new < lower || x_new > upper
            r = (rand() - 0.5)*10^-2;
            x_new = x_best + r;
        end
        f_new = f(x_new);
        if f_new > f_best
            x_best = x_new;
            f_best = f_new;
            plot(x_best,f_best,'*r');
        end
        best_hc = [best_hc, f_best];

    end

end
```

ANEXO B

```
clear("all");
f = @(x) 4 * sin(5 * pi * x + 0.5) .^ 6 .* exp(log2((x - 0.8) .^ 2));
x0 = rand();
max_iters = 100;

%Simulated Annealing
initial_temperature = 90;
cooling_rate = 0.94;


[x_best, f_best, best_sa , temp_sa] = SimulatedAnnealing(f, x0,max_iters ,0,1,
initial_temperature, cooling_rate);

 figure;
    plot(best_sa);
    title('melhor resultado');
    xlabel('best x');
    ylabel('best f');
function [x_best, f_best,best_sa,temp_sa] = SimulatedAnnealing(f, x0,
max_iters,lower,upper, initial_temperature, cooling_rate)
    figure
    hold on
    title("SA run")
    x=0:10^-5:1;
    fx = f(x);
    plot(x,fx,'b');
    it = 0;
    x_best = x0;
    f_best = f(x_best);
    best_sa = f_best;
    temp_sa = initial_temperature;
    temperature = initial_temperature;
    plot(x_best,f_best,'*r');


 % Adiciona vetores para armazenar valores de temperatura e iterações
    temp_values = [];
    iter_values = [];
     % Variáveis para armazenar o melhor resultado
    best_overall_f = [];
    best_overall_x = [];


    % Executando o algoritmo 5 vezes
    for i = 1:5

        it = 0;
        x_best = x0;
        f_best = f(x0);
        best_sa = f_best;
        temp_sa = initial_temperature;
        temperature = initial_temperature;
        while it < max_iters
            r = (rand() - 0.5)/20;
            x_new = x_best + r;
```

```
            while x_new < lower || x_new > upper
                r = (rand() - 0.5)/20;
                x_new = x_best + r;
            end
            f_new = f(x_new);
            delta_f = f_new - f_best;
            probability = exp(-abs(delta_f )/ temperature);
            if delta_f > 0
                x_best = x_new;
                f_best = f_new;
            else

                if rand() < probability
                    x_best = x_new;
                    f_best = f_new;
                end
                 figure;
            end

            temperature = temperature * cooling_rate;
            plot(x_best,f_best,'*r');
            it = it + 1 ;
            best_sa = [best_sa,f_best];
            temp_sa = [temp_sa, temperature];
            % Armazena valores de temperatura e iterações
            temp_values = [temp_values, temperature];
            iter_values = [iter_values, it];

            if f_best > best_overall_f
                best_overall_f = f_best;
                best_overall_x = x_best;
            end
        end
        fprintf('Run %d: Melhor solução x = %f com valor de função f(x) = %f\n',
i, x_best, f_best);
    end

    figure;
    plot(iter_values, temp_values, 'g'); % temperatura vs iterações
    title('Temperatura vs Iterações');
    xlabel('Iterações');
    ylabel('Temperatura');
    % Mostrar o melhor resultado


    fprintf('Melhor solução encontrada: x = %f com valor de função f(x) = %f\n',
best_overall_x, best_overall_f);
```

ANEXO C

```
clear("all");
f = @(x) 4 * sin(5 * pi * x + 0.5) .^ 6 .* exp(log2((x - 0.8) .^ 2));
x0 = rand();
max_iters = 100;
%genetic algorithm
pop_size=5;
crossover_rate = 0.8;
mutation_rate = 0.01;

[best_fit, best_solution] = GeneticAlgorithm(f, x0, max_iters, pop_size,
crossover_rate, mutation_rate);

fprintf('Best fitness: %f\n', best_fit);
fprintf('Best solution: %f %f\n', best_solution(1), best_solution(2));
```

```
function [best_fitness, best_solution] = GeneticAlgorithm(f, x0, max_iters,
pop_size, crossover_rate, mutation_rate)
    % Initialize variables
    population = init_population(pop_size, x0);
    best_fitness = f(population(1, :));
    best_solution = population(1, :);

    % Iterate until the maximum number of iterations is reached
    for i = 1:max_iters

        % Select parents
        parents = select_parents(population, best_fitness,pop_size);

        % Crossover
        offspring = crossover(parents, crossover_rate);

        % Mutation
        offspring = mutate(offspring, mutation_rate);

        % Combine with population
        new_population = [population; offspring];

        % Replace worst individuals
        population = new_population(1:pop_size, :);

        % Evaluate fitness
        fitness = f(population);

        % Update best solution and fitness
        for j = 1:pop_size
            if fitness(j) > best_fitness
                best_fitness = fitness(j);
```

```
                best_solution = population(j, :);
            end
        end
        % Ensure best_solution has at least two elements
        if size(best_solution, 2) < 2
            best_solution = [best_solution, zeros(1, 2 - size(best_solution, 2))];
        end
    end
end

function population = init_population(pop_size, x0)

    % Create empty population array
    population = zeros(pop_size, size(x0, 1));

    % Initialize each solution with random values
    for i = 1:pop_size
        population(i, :) = x0 + rand(size(x0, 1));
    end
end

function parents = select_parents(population, best_fitness,pop_size)

    % Calculate fitness vector (minimizing problem)
    fitness = best_fitness - population;
    fitness(fitness < 0) = 0;

    % Calculate cumulative fitness
    cum_fitness = cumsum(fitness);

    % Select parents using roulette wheel selection
    parents = zeros(2, size(population, 2));  % Initialize parents array with two
rows
    for i = 1:pop_size
        rand_num = rand();
        for j = 1:pop_size
            if rand_num <= cum_fitness(j)
                parents(i, :) = population(j, :);
                break;
            end
        end
    end
end

function offspring = crossover(parents, crossover_rate)

    % Randomly select crossover operator
    crossover_operator = randi(2);

    % Apply uniform crossover
    if crossover_operator == 0
        crossover_point = round(rand(1) * size(parents(1, :), 1));

        offspring(1, :) = parents(1, 1:crossover_point) + parents(2,
crossover_point+1:end);
```

```
        offspring(2, :) = parents(2, 1:crossover_point) + parents(1,
crossover_point+1:end);

    % Apply blend crossover
    else
        alpha = rand();

        offspring(1, :) = alpha * parents(1, :) + (1 - alpha) * parents(2, :);
        offspring(2, :) = (1 - alpha) * parents(1, :) + alpha * parents(2, :);
    end

end

function offspring = mutate(offspring, mutation_rate)

    % Randomly select mutation operator
    mutation_operator = randi(2);

    % Apply uniform mutation
    if mutation_operator == 0
        for i = 1:size(offspring, 1)
            for j = 1:size(offspring, 2)
                if rand() <= mutation_rate
                    offspring(i, j) = offspring(i, j) + rand();
                end
            end
        end

    % Apply Gaussian mutation
    else
        for i = 1:size(offspring, 1)
            for j = 1:size(offspring, 2)
                if rand() <= mutation_rate
                    offspring(i, j) = offspring(i, j) + randn();
                end
            end
        end
    end
end
```