UNIVERSIDADE
DE TRÁS-OS-MONTES
E ALTO DOURO

Trabalho Prático 2: Métodos de Pesquisa

Licenciatura em Engenharia Informática

Universidade de Trás-os-Montes e Alto Douro

Inteligência Artificial

Grupo 13

Paulo Moura Oliveira

Autores:

Miguel Teixeira – al78321

Rui Madureira – al78282

Vila Real, 2024

Índice

Introdução

1- Pesquisa sobre Algoritmo da Subida da Colina e *Simulated Annealing*

A otimização, uma área crucial da Inteligência Artificial, é essencial para resolver problemas complexos e desafiadores numa ampla gama de aplicações. Entre os métodos mais utilizados destacam-se o *Hill Climbing* e o *Simulated Annealing*, algoritmos que iremos abordar neste trabalho e, por sua vez, oferecem abordagens eficientes e criativas para procurar soluções ótimas.

1.1 Algoritmo da Subida da Colina

i. Definição:

O algoritmo da Subida da Colina, também conhecido como *Hill Climbing*, é uma técnica de otimização iterativa que pertence à família dos algoritmos de procura local. Este algoritmo é utilizado na ciência da computação para resolver problemas em que se pretende encontrar uma solução ótima (ou próxima do ótimo), sem necessidade de explorar exaustivamente por todo o espaço de procura. No *Hill Climbing*, a pesquisa começa num ponto inicial e "sobe" gradualmente a montanha da solução, mudando apenas para estados que melhoram a situação atual. Este processo reduz a exploração de soluções menos promissoras, mas apresenta limitações devido à possibilidade de ficar preso em mínimos locais, onde a melhoria é limitada (Pappala, 2023).

ii. Funcionamento do algoritmo:

Segundo o autor, Carneiro (2020a), o algoritmo *Hill Climbing* tem o seu funcionamento dividido em quatro fases principais, que são:

1. **Inicialização:** O algoritmo começa com uma solução inicial arbitrária, que pode ser selecionada aleatoriamente ou baseada numa estimativa inicial.
2. **Avaliação e Incremento:** Em cada iteração, o algoritmo realiza uma modificação incremental na solução atual, procurando uma solução vizinha melhor. Esta solução vizinha é obtida através de pequenas alterações na solução atual.
3. **Critério de Melhoria:** Se a nova solução encontrada for melhor que a atual, de acordo com uma função de avaliação, a solução atual é substituída pela nova.
4. **Critério de Paragem:** O processo termina quando não é possível encontrar uma solução vizinha melhor, ou seja, quando se atinge um ótimo local.

Figura 1 – Funcionamento do Algoritmo Hill Climbing: exemplo visual do processo iterativo

iii. Variantes do Algoritmo:

O algoritmo de Subida da Colina possui diversas variantes, que diferem no critério de paragem e na forma como selecionam os vizinhos. Entre as mais comuns, segundo os autores Great Learning (2020), Stochastic Hill Climbing With Random Restarts | Algorithm Afternoon, (2024) e Cloud (n.d.), destacam-se:

1. *Simple Hill Climbing* (Subida Simples):

- Avalia uma solução vizinha de cada vez e aceita-a se for melhor que a atual.
- É um método ganancioso, movendo-se apenas para soluções que melhoram o estado atual, sem considerar possíveis melhorias futuras.
  (Great Learning, 2020)

2. *Steepest-Ascent Hill Climbing* (Maior Subida):

- Examina todas as soluções vizinhas e escolhe a que oferece a maior melhoria.
- Exige mais tempo para avaliar os vizinhos, mas tende a alcançar melhores soluções em menos iterações.
  (Great Learning, 2020)

3. *Multiple Restart Hill Climbing* (Subida com Múltiplos Reinícios):

- O *Multiple Restart Hill Climbing* aumenta as chances de encontrar o ótimo global ao realizar múltiplas buscas, iniciando em pontos aleatórios diferentes.
- Esta abordagem reduz o risco de ficar preso em máximos locais, mas pode ser mais demorada devido às reinicializações frequentes.
  (*Stochastic Hill Climbing With Random Restarts* | *Algorithm Afternoon*, 2024)

4. *First-Choice Hill Climbing* (Primeira Escolha):

- Avalia as soluções vizinhas numa ordem aleatória e aceita o primeiro vizinho que melhore a solução atual.
- Pode ser mais rápido que o algoritmo *Steepest-Ascent*, mas geralmente menos preciso.
  (Cloud, n.d.)

5. *Random-Restart Hill Climbing* (Subida com Reinício Aleatório):

- O algoritmo é repetido várias vezes, iniciando sempre num ponto aleatório diferente.
- Ajuda a evitar que o algoritmo fique preso em máximos locais, aumentando as chances de encontrar o máximo global.
  (Cloud, n.d.)

iv. Aplicações do Algoritmo de Subida da Colina:

De acordo com os autores, Akhtar (2024) e Shaolin (2024), o algoritmo de Subida da Colina é amplamente aplicado em várias áreas da Inteligência Artificial e otimização, incluindo:

- **Planeamento e Agendamento:** Utilizado para criar horários e alocar recursos de forma eficiente.
- **Redes Neurais:** Aplicado na otimização e ajuste de parâmetros dos modelos.
- **Problemas de Satisfação de Restrições:** Resolvido em contextos como quebra-cabeças ou problemas lógicos complexos.

  (Akhtar, 2024)
- **Roteamento:** Empregado na otimização de rotas em redes de telecomunicações e sistemas de logística (Shaolin, 2024).

v. Síntese do algoritmo *Hill Climbing*

O algoritmo de Subida da Colina é uma técnica eficiente para encontrar ótimos locais em problemas de otimização, sendo útil especialmente quando a exploração exaustiva do espaço de procura é inviável. Permite soluções rápidas, mas apresenta limitações, como a tendência em ficar preso em ótimos locais e a dependência da solução inicial. É essencial considerar estas restrições ao aplicá-lo, principalmente ao identificar máximos e mínimos locais ou globais.

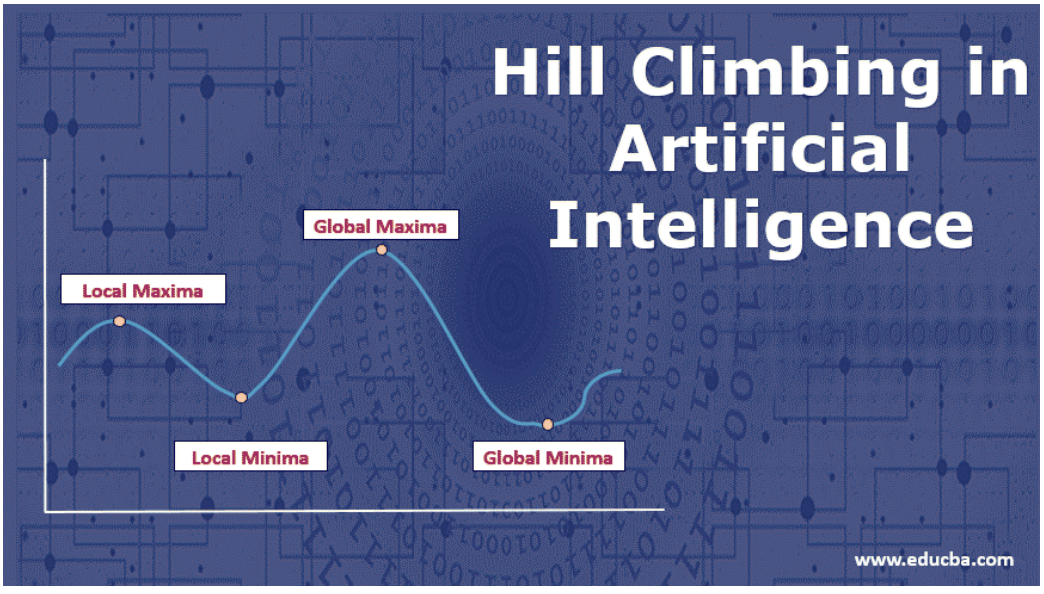

Figura 2- Algoritmo de Subida da Colina

1.2 Algoritmo *Simulated Annealing*

i. Definição:

O algoritmo *Simulated Annealing* (SA) é um método estocástico de otimização inspirado no processo de recozimento de materiais. É utilizado para encontrar soluções próximas do ótimo global em problemas complexos. O algoritmo inicia com uma solução inicial e explora o espaço de soluções aplicando pequenas alterações aleatórias. A aceitação de novas soluções é guiada por uma função de probabilidade que diminui ao longo do tempo, permitindo escapar de mínimos locais nas fases iniciais e convergir gradualmente para uma solução ideal (Guilmeau et al., 2021).

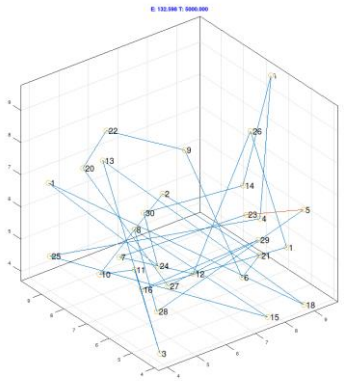

Figura 3- Representação do Algoritmo Simulated Annealing

ii. Funcionamento do algoritmo

Como referido no ponto anterior, o algoritmo SA é inspirado no processo de recozimento da metalurgia, onde um material é aquecido e depois arrefecido gradualmente, de forma a reduzir defeitos e alcançar uma estrutura mais estável. Da mesma forma, segundo o autor, Carneiro (2020a), o funcionamento do algoritmo SA desenvolve-se da seguinte maneira:

1. **Inicialização:** O algoritmo começa com uma solução inicial arbitrária e uma temperatura elevada. A temperatura controla a probabilidade de aceitar soluções piores ao longo do processo, permitindo uma exploração mais ampla do espaço de busca**.**
2. **Iteração de Busca:** Em cada iteração, uma solução vizinha é gerada a partir da solução atual, normalmente por meio de pequenas variações. Se a solução vizinha for melhor que a atual, o algoritmo faz a transição para a nova solução.
3. **Critério de Aceitação para Soluções Piores:** Se a solução vizinha for pior que a solução atual, ela pode ainda ser aceite com uma certa probabilidade, que é determinada pela função de aceitação:

$$P = e^{-\frac{\Delta E}{T}}$$

onde ΔE é a diferença entre os valores das soluções e T é a temperatura atual. Esta probabilidade decresce com o tempo, permitindo inicialmente grandes saltos no espaço de busca e, gradualmente, uma convergência mais refinada.

4. **Redução da Temperatura:** A temperatura é progressivamente reduzida de acordo com uma taxa de arrefecimento predefinida, como multiplicar a temperatura por um fator menor que 1 a cada iteração. O algoritmo continua até que a temperatura atinja um valor mínimo, momento em que as soluções vizinhas que não melhoram o resultado são quase não aceitas**.**

5. **Critério de Paragem:** O algoritmo para quando a temperatura atinge o limite inferior ou quando um número máximo de iterações é alcançado.

iii. Variantes do Algoritmo

De acordo com as ideias do autor, Xiang et al. (2023), as principais variantes do Algoritmo *Simulated Anealling* destacam-se:

1. ***Adaptive Simulated Annealing* (ASA):**

- Nesta variante, a temperatura e a taxa de arrefecimento ajustam-se dinamicamente durante o processo, com base no desempenho do algoritmo. Isso permite uma exploração mais eficiente do espaço de busca.

2. ***Parallel Simulated Annealing*:**

- Consiste na execução simultânea de múltiplas instâncias do algoritmo, explorando diferentes regiões do espaço de soluções em paralelo. Isto melhora a eficiência e aumenta a probabilidade de encontrar soluções ótima.

3. ***Hybrid Simulated Annealing*:**

- Combina o *Simulated Annealing* com outros métodos de otimização, como algoritmos genéticos ou procura local, tirando partido das vantagens de cada técnica para um desempenho global superior.

4. ***Multi-objective Simulated Annealing*:**

- Desenvolvido para problemas com múltiplos objetivos, esta variante equilibra compromissos entre diferentes critérios de otimização, gerando soluções que atendam a múltiplas metas simultaneamente.

iv. Aplicações do algoritmo *Simulated Annealing*

O *Simulated Annealing* é amplamente utilizado em problemas de otimização complexos, destacando-se, segundo o autor Kuo et al. (2022), em áreas como:

- **Otimização Combinatória**: Aplicado à organização de percursos de veículos, alocação de recursos e criação de horários.

Figura 4 - Aplicação do algoritmo Simulated Annealing em Otimização Combinatória

- **Inteligência Artificial**: Utilizado na aprendizagem de redes neuronais, incluindo a otimização de pesos e arquiteturas.
- **Engenharia**: Empregado para otimizar parâmetros em sistemas de produção e no design de layouts industriais.
- **Bioinformática**: Usado no alinhamento de sequências e na otimização de estruturas proteicas.

v. Síntese do Simulated Annealing

O *Simulated Annealing* é uma técnica eficaz para resolver problemas de otimização complexos, sobretudo quando outros métodos, como a descida de gradiente, apresentam dificuldades em superar ótimos locais. A sua capacidade de explorar soluções subótimas numa fase inicial possibilita a identificação do ótimo global. No entanto, o desempenho depende significativamente da afinação dos parâmetros. Um equilíbrio cuidadoso entre o tempo de execução e a qualidade da solução é fundamental para maximizar a sua eficácia (*Fusing Deep Learning and Sparse Coding for SAR ATR*, 2019).

2- Síntese [V1] introdutória da informação recolhida

A introdução apresentada procura sintetizar o processo de aquisição de conhecimentos e o desenvolvimento de competências no âmbito da pesquisa aprofundada efetuada e da síntese de informação relevante. Este processo de investigação tem como objetivos principais estimular a autoaprendizagem, promover a utilização eficaz de diversas ferramentas e bases de dados e contribuir para o aperfeiçoamento das competências de apresentação e comunicação dos alunos. (Paulo Moura Oliveira, 2023)

No âmbito da pesquisa sobre o algoritmo *Hill Climbing* e o *Simulated Annealing*, foram seguidas várias etapas sistemáticas, assegurando uma compreensão aprofundada sobre o tema. Numa fase inicial, realizou-se uma busca em bases de dados académicas e fontes fiáveis online, como a plataforma “b-on” (Figura 5), utilizando palavras-chave como "algoritmo de Subida da Colina", "*Hill Climbing*" e "*Simulated Annealing*". Esta metodologia permitiu-nos identificar uma ampla variedade de artigos, definições e explicações sobre os fundamentos teóricos e práticos desses algoritmos.

Figura 5- Website b-on

Após a seleção das fontes mais relevantes, as informações recolhidas foram analisadas e categorizadas. Em particular, foram investigadas as definições, os princípios de funcionamento de cada algoritmo e as suas variantes. A pesquisa também incluiu a análise das suas aplicações em áreas como inteligência artificial e otimização, o que contribuiu para contextualizar a sua relevância prática.

De forma a assegurar a precisão e a profundidade da informação, foram utilizadas fontes como o website *Great Learning*, publicações especializadas e recursos académicos. O método adotado envolveu a comparação de diferentes fontes para validar as informações, tendo em consideração diversas perspetivas e aplicações dos algoritmos. Este rigoroso processo de investigação resultou numa compreensão clara e detalhada do algoritmo de Subida da Colina, das suas variantes e das suas aplicações práticas.

Desenvolvimento

Parte II – Otimização de Funções

Em primeiro lugar, definimos a função disponibilizada no protocolo para este trabalho e implementámo-la no ficheiro denominado **f1** (Figura 6) e iremos utilizá-la ao longo do projeto.

f1.m

```
function y = f1(x)
    y = 4 * (sin(5 * pi * x + 0.5).^6) .* exp(log2((x - 0.8).^2));
end
```

Figura 6- Script que implementa a função objetivo utilizada nos algoritmos de otimização

3.1 Implementação Algoritmo *Hill Climbing*

A implementação do algoritmo de ***Hill Climbing*** tem como objetivo encontrar o máximo de uma função num intervalo pré-definido. Define-se o intervalo de procura entre [0, 1.6], e calcula-se a amplitude desse intervalo. Também, são configuradas propriedades gráficas, como a largura padrão das linhas.

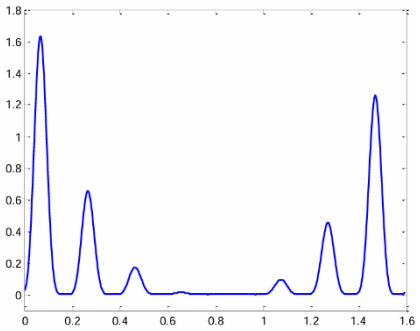

Figura 7- Gráfico da função f1

A estrutura do programa organiza a análise em três sub-gráficos apresentados numa única figura, proporcionando uma organização mais clara e uma estética melhorada. No primeiro sub-gráfico, a função objetivo é representada utilizando o comando *ezplot*, delimitada pelo intervalo de interesse. Além disso, são assinalados um ponto inicial gerado aleatoriamente é chamada a função objetivo (Figura 6) e é dado *plot* da solução inicial.

```
% Subgráfico 1: Função e Hill Climbing
subplot(3, 1, 1); % Dividindo a figura em 3 linhas e 1 coluna
ezplot('4 * (sin(5 * pi * x + 0.5).^6) .* exp(log2((x - 0.8).^2))', limits)
ylabel('f(x)')
title('Hill Climbing - Função e Soluções')
axis([0, limits(2), -0.1, 2]);
hold on;

% Gerar ponto inicial
x_current = limits(1) + rand * range;

% Chama função objetivo
F_current = f1(x_current);

% Plot da solução inicial
plot(x_current, F_current, 'or', 'MarkerSize', 6);
```

Figura 8- Plotar o gráfico do Algoritmo Hill Climbing

O algoritmo de ***Hill Climbing*** inicia com a definição de um ponto inicial gerado aleatoriamente dentro do intervalo estabelecido. A partir deste ponto, avalia-se o valor da função objetivo, sendo este valor representado graficamente por um círculo vermelho. Em seguida, o algoritmo procede com um ciclo iterativo de 300 iterações, no qual, a cada iteração, um novo ponto de teste é gerado aleatoriamente dentro do intervalo. Caso o valor da função objetivo nesse ponto de teste seja superior ao valor obtido no ponto atual, este novo ponto é aceite como o ponto de referência para a próxima iteração e é visualizado no gráfico como um círculo preto. O processo de procura local prossegue até atingir o limite máximo de iterações, conforme definido previamente.

```
xplot(1) = x_current; % Vetor guarda os valores de x
Fplot(1) = F_current; % Vetor guarda os valores de F

% Ciclo de Hill Climbing
it = 1;
num_it = 300;
while it <= num_it
    % Gerar um ponto de teste no espaço todo
    x_teste = limits(1) + rand * range;

    % Chama função objetivo
    F_teste = f1(x_teste);

    % Critério de decisão sobre se aceitamos ou não a solução nova
    if F_teste > F_current
        x_current = x_teste;
        F_current = F_teste;
        % Plot da solução nova
        plot(x_teste, F_teste, 'ok', 'MarkerSize', 6);
    end

    xplot(it) = x_current;
    Fplot(it) = F_current;

    it = it + 1;
```

Figura 9- Algoritmo Hill Climbing

O segundo sub-gráfico apresenta a evolução do valor da função objetivo ao longo das iterações, com o eixo X representando o número de iterações e o eixo Y correspondendo ao valor calculado pela função. Já o terceiro sub-gráfico ilustra a evolução do valor de x durante o processo de otimização, ou seja, a trajetória do ponto no espaço de busca ao longo das iterações. Ao terminar a execução do algoritmo, o código exibe o valor máximo encontrado para a função objetivo, fornecendo tanto a posição de x em que esse máximo ocorre, como o valor da função objetivo f(x), (Figura 6), correspondente.

```
% Subgráfico 2: Gráfico do y máximo
subplot(3, 1, 2);
It_plot = 1:1:num_it;
plot(It_plot, Fplot);
title('Gráfico do y máximo');
xlabel('Iterações');
ylabel('F');
axis([0 num_it 0 1.7]);

% Subgráfico 3: Gráfico do x máximo
subplot(3, 1, 3);
plot(It_plot, xplot);
title('Gráfico do x máximo');
xlabel('Iterações');
ylabel('x');
axis([0 num_it 0 1.7]);
```

Figura 10- Implementação dos gráficos do y e x máximo

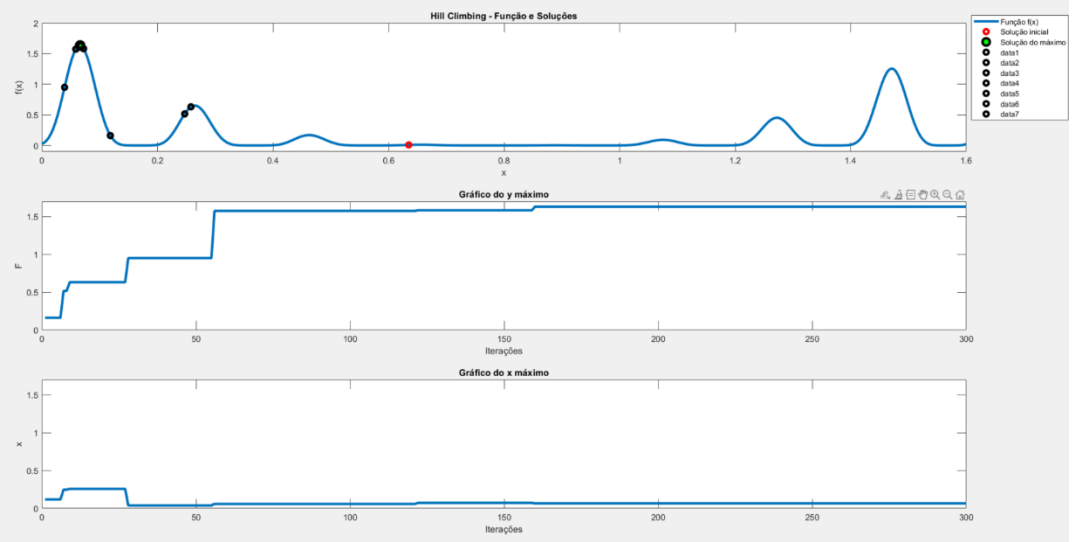

Figura 11- Resultado do Algoritmo Hill Climbing e os respetivos gráficos

3.2 Implementação Algoritmo Multiple Restart Hill Climbing

O seguinte código implementa uma variante do Algoritmo *Hill Climbing,* denominada de ***Multiple Restart Hill Climbing*** com reinicialização periódica em caso de estagnação no processo de procura, dentro do intervalo [0; 1.6], (Figura 12). A execução do código divide-se em diferentes etapas, com gráficos que permitem a visualização dos resultados obtidos

```
limits = [0, 1.6];
range = limits(2) - limits(1);
set(0, 'defaultlinelinewidth', 3);
```

Figura 12- Definição dos limites do gráfico e a largura das linhas dos gráficos

Inicialmente, são definidos os limites da função e configurado o espaço de procura. Em seguida, no primeiro sub-gráfico, é plotada a função objetivo f(x), representando visualmente o seu comportamento ao longo do intervalo. Um ponto inicial é gerado aleatoriamente dentro do intervalo, sendo avaliado pela função objetivo, ilustrado através de um círculo vermelho. O máximo global é, também, destacado no gráfico como um círculo verde para fins de comparação.

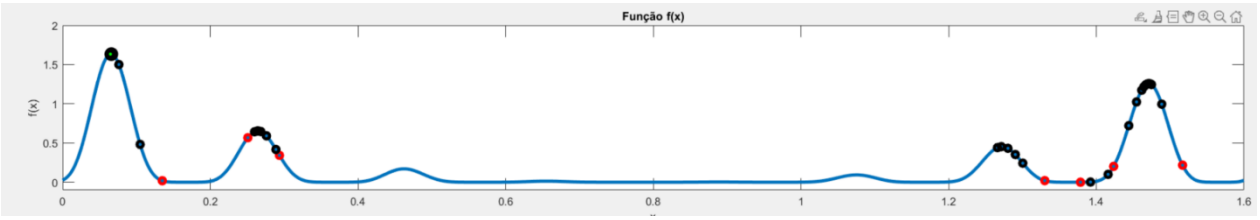

Figura 13- Multiple Restart Hill Climbing

O algoritmo é executado ao longo de 300 iterações,(Figura 14), com a cada iteração é gerado um novo ponto de teste numa vizinhança definida pelo parâmetro step_size, que corresponde a 2% da amplitude do espaço de busca, (Figura 15). Se o valor da função no ponto de teste for superior ao valor atual, este ponto é aceite como o novo ponto de referência, sendo armazenado, (Figura 16) e plotado no gráfico. Caso contrário, o contador de iterações sem melhoria é incrementado. Para evitar estagnações prolongadas, é implementada uma lógica de reinicialização: quando o número de iterações sem melhoria excede 25, o algoritmo reinicia com um novo ponto aleatório dentro do espaço de busca, redefinindo o contador, (Figura 17).

```
it = 1;
num_it = 300;
max_no_improvement = 25;
no_improvement_count = 0;
step_size = 0.02 * range;
```

Figura 14- Definição Variáveis utilizadas no algoritmo

```
x_teste = x_current + (rand * 2 - 1) * step_size;
x_teste = max(min(x_teste, limits(2)), limits(1));
```

Figura 15- Gerar o ponto de teste

```
F_teste = f1(x_teste);
if F_teste > F_current
    x_current = x_teste;
    F_current = F_teste;
    plot(x_teste, F_teste, 'ok', 'MarkerSize', 6);
    no_improvement_count = 0;
else
    no_improvement_count = no_improvement_count + 1;
end

% Armazenamento dos valores da iteração
xplot(it) = x_current;
Fplot(it) = F_current;
```

Figura 16- Verificação de melhoria

```
if no_improvement_count >= max_no_improvement
    x_current = limits(1) + rand * range;
    F_current = f1(x_current);
    no_improvement_count = 0;
    plot(x_current, F_current, 'or', 'MarkerSize', 6);
end

it = it + 1;
```

Figura 17- Verificação para reinicialização MRHC

Os resultados obtidos ao longo das iterações são apresentados nos sub-gráficos seguintes, (Figura 21). Assim como o *Algoritmo Hill Climbing*, o segundo sub-gráfico ilustra a evolução do valor máximo da função f(x) ao longo das iterações, enquanto o terceiro sub-gráfico mostra a evolução do valor de x ao longo das iterações, (Figura 21). No gráfico principal, os diferentes círculos destacam elementos importantes do algoritmo: a bola verde, (Figura 18) representa o máximo global conhecido, a bola com contorno vermelho, (Figura 19) indica a solução inicial aleatória, e as bolas com contorno preto , (Figura 20) mostram os pontos de melhoria encontrados ao longo do algoritmo.

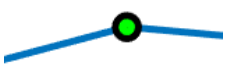

Figura 18- Máximo Global

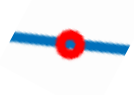

Figura 19- Solução Inicial aleatória

Figura 20- Pontos de Melhoria

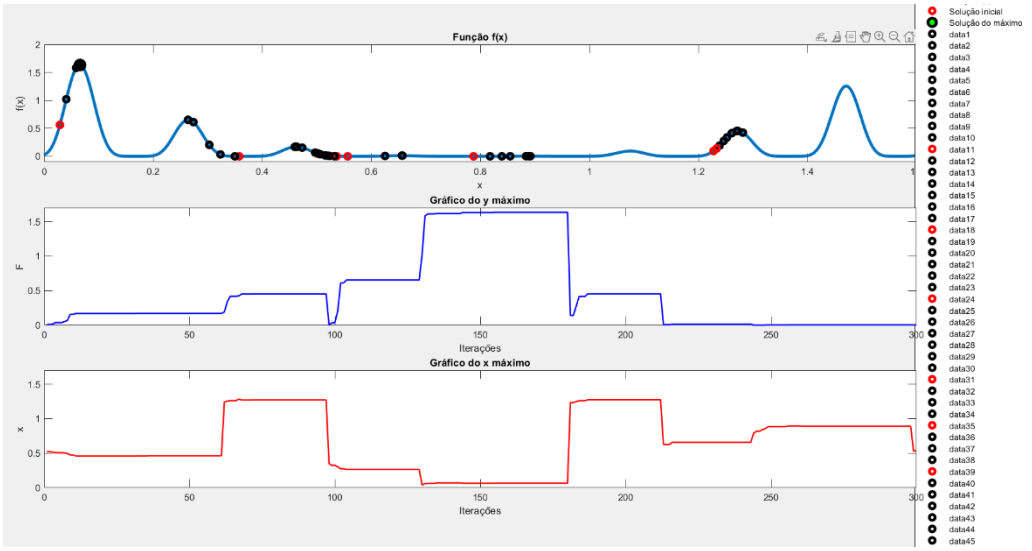

Figura 21- Resultado algoritmo MRHC e os respetivos gráficos

3.3 Implementação Algoritmo Simulated Annealing (SA)

No desenvolvimento deste ponto do protocolo, decidimos, dada a complexidade do algoritmo, dividir a implementação em dois ficheiros distintos para a execução do *Simulated Annealing*. Assim, criámos o ficheiro ***simulated_annealing.m***, responsável pelos cálculos fundamentais de otimização. Concebemos, também, o script de execução, denominado de ***run_simulated_annealing.m***, que organiza várias execuções, interpreta os resultados e apresenta análises gráficas, que permitem uma melhor compreensão do comportamento do algoritmo.

Relativamente ao script ***simulated_annealing.m***, o algoritmo começa com uma temperatura elevada, (Figura 22), que vai diminuindo a cada iteração. Esta temperatura controla a probabilidade de aceitar soluções piores, o que ajuda a explorar o espaço de procura e evita ficar preso em máximos locais.

A cada iteração, o algoritmo gera um novo ponto x aleatório próximo do ponto atual e calcula o valor da função f1 nesse novo ponto. Calcula também a diferença de energia (‘dE’) entre o valor da função no novo ponto e no ponto atual. Se o valor for melhor, o ponto é aceite. No entanto, mesmo que a solução seja pior, esta pode ainda ser aceite com uma probabilidade definida pela fórmula de *Boltzmann*, dependente de ‘dE’ e da temperatura atual. Este comportamento é implementado no ciclo *for* que itera sobre ‘nRep’ , onde se calcula ‘dE’ e a probabilidade de aceitação ‘prob’, sendo estas verificadas pela condição de aceitação, (Figura 23).

```
T = 100;
nRep = 500;
max_iter = 300;
x_min = 0;
x_max = 1.6;
delta_x = 0.02 * (x_max - x_min);
```

Figura 22- Variáveis utilizadas no simulated_annealing.m

```
for iter = 1:max_iter
    for i = 1:nRep
        x_new = x_current + delta_x * (2 * rand - 1);
        x_new = max(x_min, min(x_max, x_new));
        f_new = f1(x_new);
        dE = f_new - f_current;
        prob = exp(dE / T);
        if dE > 0 || prob > rand
            x_current = x_new;
            f_current = f_new;
        end
        if f_current > f_best
            x_best = x_current;
            f_best = f_current;
        end
        x_history(step) = x_current;
        f_history(step) = f_current;
        probabilities(step) = prob;
        iteracoes(step) = (iter - 1) * nRep + i;
        step = step + 1;
    end
    temperatures(iter) = T;
    T = T * alfa;
end
```

Figura 23- Algoritmo Simulated Annealing

```
x_history = x_history(1:step-1);
f_history = f_history(1:step-1);
probabilities = probabilities(1:step-1);
iteracoes = iteracoes(1:step-1);
```

Figura 24- Armazenamento do histórico de todas as iterações

O processo continua por um número fixo de iterações, durante as quais o algoritmo regista os valores de x, f(x), a temperatura e as probabilidades calculadas. No final, devolve o melhor valor de x e o respetivo valor da função, juntamente com o histórico de todas as iterações, (Figura 24).

Por sua vez, o script ***run_simulated_annealing.m,*** expande o uso do *Simulated Annealing*, onde decidimos analisar o impacto de diferentes valores do alfa (0.86, 0.90 e 0.94) no comportamento do algoritmo. Inicialmente, o algoritmo é executado com alfa = 0.90, retornando o melhor valor de x (x_best) e o respetivo valor da função objetivo (f_best). Esses resultados são exibidos na consola e no primeiro gráfico, que representa a função objetivo f1(x) no intervalo [0, 1.6], destacando visualmente o ponto de máximo encontrado, (Figura 25). Esta abordagem valida a eficácia do algoritmo na identificação de soluções próximas do máximo global da função.

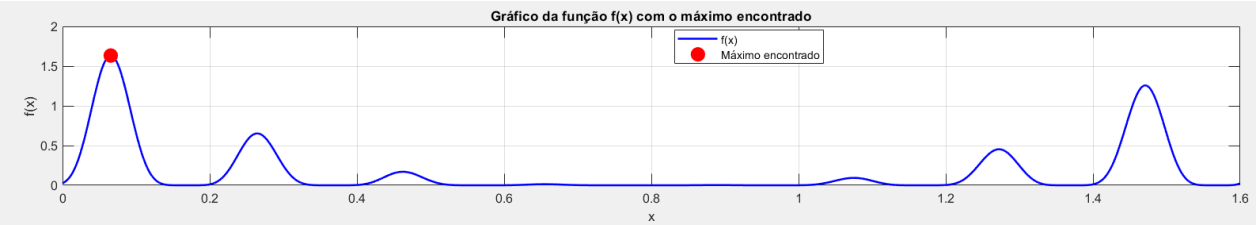

Figura 25- Máximo encontrado recorrendo ao Algoritmo Simulated Annealing

O código avalia como a temperatura evolui ao longo das iterações para diferentes valores de alfa, utilizando a função auxiliar **calculate_temperatures**. Valores maiores de alfa promovem um resfriamento mais lento, permitindo maior exploração, enquanto valores menores aceleram o resfriamento, focando numa exploração mais localizada, (Figura 26). Simultaneamente, a probabilidade de aceitar novas soluções, baseada no modelo de Boltzmann, é calculada através da função **calculate_boltzmann_probabilities**. Esta probabilidade diminui à medida que a temperatura baixa, refletindo a transição do algoritmo de uma fase exploratória para uma fase de refinamento, (Figura 27). Os gráficos mostram claramente o impacto de alfa na temperatura e na probabilidade de aceitação, ilustrando como estes fatores influenciam a eficiência do algoritmo.

A relação com o código anterior reside na utilização do mesmo algoritmo de *Simulated Annealing* para encontrar o máximo da função. Contudo, enquanto o código anterior implementa a lógica básica e avalia uma única execução, este aprofunda a análise, explorando como diferentes valores de alfa afetam o resfriamento, a probabilidade de aceitação e, consequentemente, a eficiência do algoritmo na exploração do espaço de soluções. Para além disso, é aqui que são plotados todos os gráficos relativos ao Algoritmo *Simulated Annealing,* (Figuras 25, 26, 27)

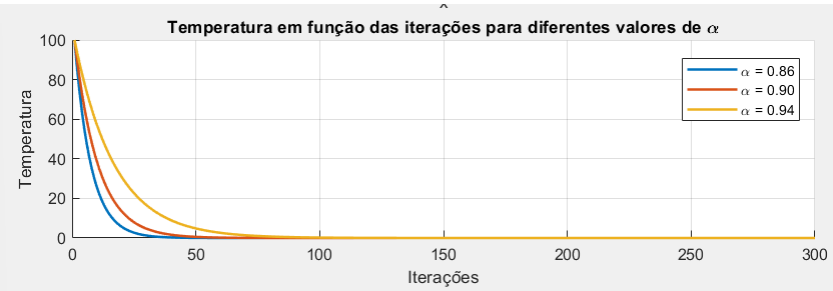

Figura 26- Temperatura em função das iterações para diferentes valores de alfa

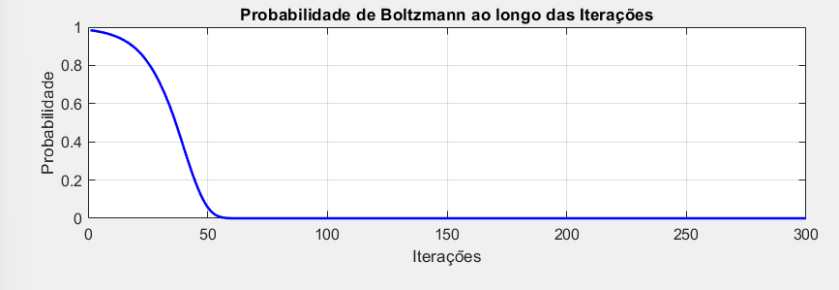

Figura 27- Probabilidade de Boltzmann ao longo das Iterações

Parte III – Problema do Caixeiro Viajante

Assim como no *Simulated Annealing*, no Problema do Caixeiro Viajante, decidimos implementar dois scripts para melhorar a organização e a estética do código, intitulados de **cidadesEscolhidas** e **TSP**.

O primeiro código, **cidadesEscolhidas**, é uma função que define as coordenadas geográficas e os nomes das cidades portuguesas em diferentes conjuntos, dependendo da opção selecionada pelo utilizador. A opção ‘a’ corresponde a 14 cidades, ‘b’ a 20 cidades e ‘c’ a 30 cidades, onde cada cidade é representada pela sua latitude e longitude, e os nomes são armazenados numa lista separada. Assim, a função facilita a reutilização e integração de diferentes conjuntos de dados noutros algoritmos.

```
function [cidades, nomesCidades] = cidadesEscolhidas(opcao)
    switch opcao
        case 'a'
            % 14 cidades
            cidades = [
                41.8167, -6.7500;  % Bragança
                41.3000, -7.7500;  % Vila Real
                41.7333, -7.4667;  % Chaves
                41.7000, -8.8333;  % Viana do Castelo
                41.5500, -8.4333;  % Braga
                40.6333, -8.6500;  % Aveiro
                41.1833, -8.6000;  % Porto
                40.6500, -7.9167;  % Viseu
                41.1000, -7.8167;  % Lamego
                40.5667, -8.4500;  % Águeda
                41.1667, -7.7833;  % Peso da Régua
                41.4500, -8.3000;  % Guimarães
                42.0333, -8.6333;  % Valença do Minho
                41.5333, -8.6167;  % Barcelos
            ];
            nomesCidades = {
                'Bragança', 'Vila Real', 'Chaves', 'Viana do Castelo', 'Braga', ...
                'Aveiro', 'Porto', 'Viseu', 'Lamego', 'Águeda', 'Peso da Régua', ...
                'Guimarães', 'Valença do Minho', 'Barcelos'
            };
```

Figura 28- Script cidadesEscolhidas caso o utilizador selecione 14 cidades

O segundo código implementa a resolução do problema do Caixeiro Viajante (TSP), utilizando o algoritmo *Simulated Annealing*. Inicialmente, o utilizador escolhe uma das opções de cidades, e a função **cidadesEscolhidas** é chamada para carregar as respetivas coordenadas e nomes, (Figura 29). Em seguida, o programa calcula uma matriz de distâncias entre todas as cidades. Para isso, recorremos a várias fontes e decidimos utilizar uma fórmula baseada na esfera terrestre para determinar a distância geográfica entre duas localizações, (Figura 30).

```
disp('Por quantas cidades Portuguesas o Caixeiro Viajante passa?');
disp('a) 14 cidades');
disp('b) 20 cidades');
disp('c) 30 cidades');
resposta = input('Escolha a opção (a, b ou c): ', 's');
```

Figura 29- Input de quantas cidades o utilizador quer analisar

```
distancia = @(lat1, lon1, lat2, lon2) ...
    6371 * acos(sin(deg2rad(lat1)) .* sin(deg2rad(lat2)) + ...
    cos(deg2rad(lat1)) .* cos(deg2rad(lat2)) .* cos(deg2rad(lon2 - lon1)));
```

Figura 30- Fórmula para determinar distância entre duas localidades

Com a matriz de distâncias definida e a função de custo para um dado percurso, o algoritmo *Simulated Annealing,* utilizado nos códigos ***run_simulated_annealing.m*** e ***simulated_annealing.m,*** é executado para encontrar o percurso ótimo que minimiza a distância total. Este método utiliza um processo iterativo de resfriamento gradual, controlado por parâmetros como a temperatura inicial, a taxa de resfriamento (*alpha*) e o número de iterações por temperatura, (Figura 31). Durante cada iteração, são geradas novas soluções (alterando a ordem das cidades no percurso), e a aceitação dessas soluções é decidida com base na diferença de custo e na probabilidade de *Boltzmann,*(Figura 32). Com a diminuição da temperatura, o algoritmo equilibra a exploração de soluções diversificadas e o refinamento para valores próximos ao ótimo, ajustando sua probabilidade de aceitar soluções subótimas.

No final, o código apresenta a melhor rota encontrada e o respetivo custo total. Também gera gráficos para analisar o comportamento do algoritmo, incluindo a evolução da temperatura e a probabilidade de *Boltzmann* ao longo das iterações. Por fim, a rota otimizada é plotada num mapa geográfico, com as cidades marcadas e os seus nomes exibidos, (Figuras 33, 34, 35).

```
T = 1000;
T_min = 1e-3;
alpha = 0.995;
iter = 100;
```

Figura 31- Parâmetros definidos

```
if deltaE < 0
    probabilidade = 1;
else
    probabilidade = exp(-deltaE / T);
end


    probabilidades = [probabilidades; probabilidade];


if novoCusto < custoAtual || rand < probabilidade
    rotaAtual = novaRota;
end
```

Figura 32- Probabilidade de Boltzmann

Por fim, optamos por implementar uma funcionalidade que permite armazenar os melhores resultados num ficheiro denominado **melhores_resultados.txt**, (Figura 36). Este ficheiro regista, tal como o nome indica, as melhores soluções obtidas para diferentes quantidades de cidades, guardando sempre a rota com o menor custo possível. Sempre que o programa gerar uma rota com um custo inferior ao que se encontra no arquivo, esta será substituída pela nova rota gerada. Este processo aplica-se de forma contínua, quer para catorze, quer para vinte ou trinta cidades, garantindo assim que o ficheiro mantém sempre os melhores resultados encontrados.

Resultados do Problema do Caixeiro Viajante

- 14 cidades


```
Por quantas cidades Portuguesas o Caixeiro Viajante passa?
a) 14 cidades
b) 20 cidades
c) 30 cidades
Escolha a opção (a, b ou c): a
Melhor rota encontrada entre as 14 cidades:
Bragança -> Chaves -> Valença do Minho -> Viana do Castelo -> Barcelos -> Braga
Guimarães -> Porto -> Aveiro -> Águeda -> Viseu -> Lamego
Peso da Régua -> Vila Real
Custo da melhor rota = 599.2353
```

Figura 33- Resultado do Problema do Caixeiro viajante para 14 cidades

- 20 cidades


```
Melhor rota encontrada entre as 20 cidades:
Setúbal -> Sagres -> Faro -> Tavira -> Évora -> Portalegre
Guarda -> Viseu -> Lamego -> Vila Real -> Bragança -> Chaves
Viana do Castelo -> Braga -> Guimarães -> Porto -> Aveiro -> Coimbra
Santarém -> Lisboa
Custo da melhor rota = 1500.3198
```

Figura 34 - Resultado do Problema do Caixeiro viajante para 20 cidades

- 30 cidades


```
Melhor rota encontrada entre as 30 cidades:
Tomar -> Santarém -> Sintra -> Lisboa -> Setúbal -> Sines
Sagres -> Faro -> Tavira -> Beja -> Évora -> Elvas
Portalegre -> Castelo Branco -> Covilhã -> Guarda -> Miranda do Douro -> Bragança
Chaves -> Vila Real -> Lamego -> Guimarães -> Braga -> Viana do Castelo
Porto -> Aveiro -> Águeda -> Viseu -> Coimbra -> Leiria
Custo da melhor rota = 1678.5279
```

Figura 35- Resultado do Problema do Caixeiro viajante para 30 cidades

```
melhores_resultados.txt
Melhor resultado para 20 cidades:
Rota: Coimbra -> Portalegre -> Santarém -> Lisboa -> Setúbal -> Sagres -> Faro -> Tavira -> Évora -> Guarda -> Bragança -> Chaves -> Vila Real -> Lamego -> Viseu -> Guima
Custo: 1674.67 km
----------------------------------------
Melhor resultado para 30 cidades:
Rota: Tomar -> Évora -> Beja -> Tavira -> Faro -> Sagres -> Sines -> Setúbal -> Lisboa -> Sintra -> Santarém -> Elvas -> Portalegre -> Castelo Branco -> Covilhã -> Águeda
Custo: 1895.58 km
----------------------------------------
Melhor resultado para 14 cidades:
Rota: Braga -> Barcelos -> Viana do Castelo -> Valença do Minho -> Chaves -> Bragança -> Vila Real -> Peso da Régua -> Lamego -> Viseu -> Águeda -> Aveiro -> Porto -> Gui
Custo: 599.24 km
----------------------------------------
```

Figura 36- Ficheiro melhores_resultados.txt

Conclusão

A execução deste trabalho proporcionou uma análise aprofundada de técnicas de otimização, como os algoritmos *Hill Climbing* e *Simulated Annealing*, explorando os seus princípios de funcionamento, variantes e aplicações práticas. A implementação destes métodos permitiu compreender os seus pontos fortes e limitações, especialmente em problemas complexos como a otimização de funções e o Problema do Caixeiro Viajante, onde soluções exatas não são viáveis devido à elevada dimensão do espaço de procura.

Um dos grandes contributos deste projeto foi a consolidação das nossas competências no uso do *MATLAB*, uma ferramenta poderosa que facilitou a implementação e análise dos algoritmos desenvolvidos. Desde a estruturação do código até à criação de gráficos para interpretar resultados, o trabalho reforçou o nosso domínio técnico e a capacidade de aplicar métodos computacionais a cenários práticos.

Os resultados obtidos demonstraram que ambos os algoritmos têm uma aplicabilidade significativa em problemas de otimização, mas com características que os tornam mais adequados a diferentes contextos. Por exemplo, enquanto o *Hill Climbing* é mais rápido, pode ficar preso em ótimos locais; já o *Simulated Annealing*, embora mais lento, mostrou-se eficaz em superar mínimos e máximos locais, alcançando soluções mais robustas.

Este trabalho não só aprofundou os conhecimentos em inteligência artificial e técnicas de otimização, mas também estimulou o pensamento crítico na escolha de abordagens e na análise de resultados. Assim, conclui-se que o projeto foi essencial para desenvolver uma visão abrangente e prática sobre a resolução de problemas complexos, promovendo aprendizagens que serão úteis em futuros desafios académicos e profissionais.

Bibliografia

Akhtar, Z. (2024, July 16). Hill climbing algorithm in artificial intelligence. *DatabaseTown*. Retrieved November 30, 2024, from https://databasetown.com/Hill-climbing-algorithm-in-artificial-intelligence/#use-cases-and-applications

Carneiro, A. L. C. (2020a, June 14). Algoritmos de otimização: Hill Climbing e Simulated Annealing. *Medium*. https://medium.com/data-hackers/algoritmos-deotimiza%C3%A7%C3%A3o-hill-climbing-e-simulated-annealing-3803061f66f0

Cloud, C. (n.d.). *Essential guide to hill climbing in Artificial intelligence*. Cyfuture Cloud. https://cyfuture.cloud/kb/general/introduction-to-Hill-climbing-in-artificial-intelligence

Fusing deep learning and sparse coding for SAR ATR. (2019, April 2). IEEE Journals & Magazine | IEEE Xplore. Retrieved November 30, 2024, from https://ieeexplore.ieee.org/document/8432096

Guilmeau, T., Chouzenoux, E., & Elvira, V. (2021, July 1). *Simulated annealing: a review and a new scheme*. Retrieved November 30, 2024, from https://inria.hal.science/hal-03275401

Great Learning. (2020, May 22). An introduction to hill climbing algorithm in AI (Artificial Intelligence). *Great Learning Blog: Free Resources What Matters to Shape Your Career!* https://www.mygreatlearning.com/blog/an-introduction-to-Hill-climbing-algorithm/

Kuo, C. L., Kuruoglu, E. E., & Chan, W. K. V. (2022). Neural network structure optimization by simulated annealing. *Entropy*, *24*(3), 348. https://doi.org/10.3390/e24030348

Pappala, D. S. (7 de 2023). *Hill-Climbing Search Artificial Intelligence*, p. 8. https://www.researchgate.net/publication/378679524_Hill-Climbing-Search_Artificial_Intelligence

Paulo Moura Oliveira. (2023, March 8). *Trabalho de Síntese* [Video]. YouTube. https://www.youtube.com/watch?v=t_OZbolXY9s

Stochastic hill climbing with random restarts | Algorithm Afternoon. (2024, April 16). https://algorithmafternoon.com/stochastic/stochastic_hill_climbing_with_random_restarts/

Xiang, J., Zhang, Y., Cao, X., & Zhou, Z. (2023). An Improved Multi-Objective Hybrid Genetic-Simulated Annealing Algorithm for AGV Scheduling under Composite Operation Mode. *Computers, Materials & Continua/Computers, Materials & Continua (Print)*, *77*(3), 3443–3466. https://doi.org/10.32604/cmc.2023.045120