P.PORTO

ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO

Análise Algorítmica e Otimização

Uncapacitated Facility Location Problem

José Henrique Noronha Oliveira 8220343@estg.ipp.pt

Lucas Teixeira Fernandes 8220297@estg.ipp.pt

Rodrigo de Queirós Alves 8220816@estg.ipp.pt

João Pedro Salgado Pereira 8220102@estg.ipp.pt

Índice

Índice de Figuras

Índice de Tabelas

Lista de Siglas e Acrónimos

UFLP Uncapacitated Facility Location Problem (Problema de Localização de Instalações sem Restrições de Capacidade)

SA Simulated Annealing

BCS Binary Crow Search

CSA Crow Search Algorithm

GRASP Greedy Randomized Adaptive Search

HC Hill Climbing

RCL Lista Restrita de Candidatos

1. Introdução

O presente relatório aborda o Problema de Localização de Instalações sem Restrições de Capacidade (UFLP), um problema significativo em pesquisa operacional e otimização combinatória. O UFLP é essencial para setores como logística, planeamento de redes e gestão da cadeia de suprimentos, onde decisões eficazes de localização podem gerar economias substanciais e melhorias operacionais. O trabalho proposto visa explorar e desenvolver métodos heurísticos para resolver o UFLP, destacando-se pela sua capacidade de fornecer soluções aproximadas, mas computacionalmente viáveis e eficazes, especialmente para grandes instâncias onde métodos exatos são impraticáveis. A implementação e comparação de três métodos heurísticos - Binary Crow Search, Hill Climbing, GRASP, Simulated Annealing - constituem o cerne deste estudo, focando na eficiência e qualidade das soluções obtidas.

1.1 Contextualização

O UFLP é um problema clássico em pesquisa operacional e otimização combinatória, que envolve a seleção de localizações para um conjunto de instalações, de forma a minimizar os custos totais, que incluem custos fixos de abertura das instalações e custos variáveis de transporte dos clientes para as instalações. Esse problema é de grande relevância em setores como logística, planeamento de redes e gestão da cadeia de suprimentos, onde decisões eficazes de localização podem resultar em economias substanciais e melhorias operacionais.

1.2 Apresentação do Caso de Estudo

A proposta deste trabalho surge da necessidade de explorar e desenvolver métodos heurísticos para resolver o UFLP. Os principais motivos incluem a complexidade do problema, que torna impraticáveis métodos exatos para grandes instâncias, e a pesquisa por soluções que, embora aproximadas, sejam computacionalmente viáveis e eficazes. Este estudo tem como objetivos a criação de algoritmos heurísticos para o UFLP e a avaliação do seu desempenho em termos de eficiência e qualidade das soluções obtidas.

1.3 Motivação e Objetivos

A motivação para este trabalho reside na importância prática do UFLP em diversas indústrias e na necessidade de desenvolver algoritmos que possam fornecer soluções rápidas e de alta qualidade. A complexidade do problema e a necessidade de otimização em tempo real fazem com que métodos heurísticos sejam uma escolha natural. Os objetivos específicos deste trabalho são:

- Utilizar métodos heurísticos no desenvolvimento de algoritmos para a resolução de instâncias do UFLP.
- Aplicar conhecimentos sobre análise algorítmica para avaliar o desempenho dos algoritmos implementados.
- Especificamente, implementar e comparar quatro métodos heurísticos: Binary Crow Search, Hill Climbing, GRASP, Simulated Annealing.

1.4 Estrutura do Relatório

Capítulo 1: Introdução ao problema de Localização de Instalações sem Restrições de Capacidade

- 1.1 Contextualização do problema apresentado.
- 1.2 Apresentação do caso de estudo.
- 1.3 Motivação e objetivos do estudo.
- 1.4 Estrutura do relatório.

Capítulo 2: Pesquisa bibliográfica sobre o Problema de Localização de Instalações sem Restrições de Capacidade

- 2.1 Métodos Heurísticos: Discussão sobre diferentes abordagens heurísticas aplicáveis ao UFLP.
- 2.2 Estado da Arte do UFLP: Revisão da literatura existente sobre o UFLP e suas soluções.

Capítulo 3: Implementação de Algoritmos para a Resolução do Problema de Localização de Instalações sem Restrições de Capacidade

- 3.1 Algoritmo Binary Crow Search: Descrição, desenvolvimento e implementação.
- 3.2 Algoritmo Simulated Annealing: Descrição, desenvolvimento e implementação.

- 3.3 Algoritmo GRASP: Descrição, desenvolvimento e implementação.
- 3.4 Algoritmo Hill Climbing: Descrição, desenvolvimento e implementação.

Capítulo 4: Análise de Desempenho dos Algoritmos Implementados

- 4.1 Tempos de Execução: Avaliação dos tempos de execução dos algoritmos.
- 4.2 Qualidade das Soluções Obtidas: Análise da qualidade das soluções encontradas.
- 4.3 Comparação dos Algoritmos Implementados: Comparação entre as diferentes abordagens heurísticas.

Capítulo 5: Conclusões e Trabalhos Futuros

- Resumo dos principais achados do trabalho.
- Discussão sobre os pontos fortes e fracos dos algoritmos implementados.
- Sugestões para futuras pesquisas e possíveis melhorias nos algoritmos desenvolvidos.

2. Pesquisa bibliográfica sobre o Problema de Localização de Instalações sem Restrições de Capacidade

O Problema de Localização de Instalações sem Restrições de Capacidade (UFLP) é uma questão chave na logística e na gestão de operações, focando na determinação das melhores localizações para instalações, como armazéns ou centros de distribuição, sem considerar limitações de capacidade. O objetivo principal é minimizar os custos totais, que podem incluir transporte, construção e operação. Este problema tem ampla aplicação em setores como o retalho, serviços públicos e manufatura.

Duas abordagens destacadas para resolver o UFLP são a Filter-and-Fan e a Tabu Search. A abordagem Filter-and-Fan, conforme discutido por Greistorfer e Rego em *"Asimple filter-and-fan approach to the facility location problem"*, combina um processo de filtragem inicial para reduzir o espaço de pesquisa, eliminando soluções subótimas, seguido por uma expansão sistemática para encontrar soluções ótimas. Este método é eficaz pela sua simplicidade e capacidade de rapidamente identificar boas soluções.

Por outro lado, a Tabu Search, abordada por Michel e Van Hentenryck no seu artigo, *"A simple tabu search for warehouse location"*, utiliza uma estratégia de optimização baseada em heurísticas que evita revisitar soluções já exploradas, armazenando movimentos proibidos numa lista tabu. Esta técnica é valiosa para explorar intensivamente o espaço de soluções, promovendo uma pesquisa diversificada e eficiente.

Ambas as metodologias oferecem maneiras robustas de abordar o UFLP, auxiliando empresas a tomar decisões informadas sobre a localização das suas instalações, o que pode resultar em significativas economias de custo e melhorias operacionais.

O Problema de Localização de Facilidades Não Capacitado (UFLP) pode ser descrito matematicamente da seguinte forma:

- **Variáveis de decisão:**
  - x_i: variável binária que indica se o armazém i está aberto (x_i = 1) ou fechado (x_i = 0).
  - y_{ij}: variável binária que indica se o cliente j está sendo atendido pelo armazém (y_{ij} = 1) ou não(y_{ij} = 0).
- **Função objetivo:** Minimizar o custo total C:

$$C = \sum_{i \in W} f_i x_i + \sum_{i \in C} \sum_{i \in W} c_{ij} y_{ij}$$

onde:

- W é o conjunto de armazéns.
- C é o conjunto de clientes.
- f_i é o custo fixo de abrir o armazém.
- c_{ij} é o custo de atender o cliente já pelo armazém i.

- **Restrições:**
  - Cada cliente deve ser atendido por exatamente um armazém:

$$\sum_{i \in W} y_{ij} = 1 \; \forall j \in C$$

  - Um cliente só pode ser atendido por um armazém que esteja aberto:

$$y_{ij} \leq x_i \; \forall i \in W, \forall j \in C$$

  - As variáveis x_i e y_{ij} são binárias:

$$x_i \in \{0, 1\} \; \forall i \in W$$

$$y_{ij} \in \{0, 1\} \; \forall i \in W, \forall j \in C$$

3. Implementação de algoritmos para a resolução do Problema de Localização de Instalações sem Restrições de Capacidade

3.1 Algoritmo Binary Crow Search

O algoritmo Binary Crow Search (BCS) é uma adaptação do algoritmo Crow Search, inspirado no comportamento social dos corvos. Este algoritmo é utilizado para resolver problemas de otimização combinatória, como o Problema de Localização de Instalações sem Restrições de Capacidade. No BCS, a pesquisa ocorre num espaço binário, onde cada posição representa uma solução potencial para o problema.

3.1.1 História

O algoritmo Crow Search (CS) foi desenvolvido a partir da observação do comportamento social e cognitivo dos corvos. Estes pássaros são reconhecidos pela sua inteligência, memória robusta e capacidade de resolver problemas complexos, o que inclui lembrar-se de locais onde esconderam alimento e a habilidade de roubar de forma despercebida o alimento de outros corvos. Este comportamento serviu como base para a criação do algoritmo Crow Search.

O CS foi inicialmente concebido para resolver problemas de otimização contínua. No entanto, devido à sua flexibilidade e eficiência, o algoritmo foi posteriormente adaptado para lidar com problemas de otimização combinatória, resultando na criação do Binary Crow Search. O BCS opera num espaço de pesquisa binário, onde cada posição representa uma solução potencial, sendo particularmente eficaz em problemas onde as soluções são naturalmente representadas de forma binária.

Uma aplicação notável do BCS é no problema de Localização de Instalações sem Restrições de Capacidade. A capacidade do BCS de explorar e explorar eficazmente o espaço de soluções torna-o uma ferramenta valiosa para abordar este problema.

3.1.2. Introdução do método relativamente ao UFLP

O BCS é uma ferramenta altamente eficaz para resolver o Problema de Localização de Instalações sem Restrições de Capacidade. Este problema envolve a determinação das melhores localizações para um conjunto de instalações de forma a minimizar os custos totais, que podem incluir custos de construção, operação e transporte, entre outros.

No contexto do UFLP, cada solução potencial pode ser representada como um vetor binário, onde cada posição no vetor indica se uma instalação está presente (1) ou ausente (0) num determinado local. O BCS utiliza uma população de "corvos" (soluções) que "voam" através deste espaço de pesquisa binário. Cada corvo possui uma memória que armazena a melhor solução encontrada até o momento.

A eficácia do BCS no UFLP deve-se à sua capacidade de equilibrar a exploração e a exploração do espaço de soluções. A exploração refere-se à procura de novas soluções em áreas ainda não investigadas do espaço de pesquisa, enquanto a exploração foca em refinar soluções em áreas que já demonstraram potencial. Este equilíbrio é crucial para evitar que o algoritmo fique preso em soluções locais subótimas e para garantir que o espaço de soluções seja adequadamente coberto.

3.1.3. Descrição do Algoritmo

O BCS utiliza uma população de corvos (soluções) que voam num espaço de pesquisa binário, onde cada posição representa uma solução potencial. Cada corvo tem uma memória que armazena a melhor solução encontrada. A pesquisa dos corvos é um equilíbrio entre a exploração de novas áreas e a exploração de áreas conhecidas.

3.1.3.1 Passo a Passo

1. **Inicialização**
    - Definir os parâmetros do algoritmo: número de corvos (N), probabilidade de descoberta (AP), número máximo de iterações (MAX_ITER).
    - Iniciar aleatoriamente a posição dos corvos no espaço de pesquisa binário.
    - Iniciar a memória dos corvos com as suas posições iniciais.
2. **Iteração**
    - Para cada corvo:
        - Gerar uma posição candidata baseada na memória e na probabilidade de descoberta.
        - Avaliar a solução na posição candidata.

- Atualizar a posição e a memória do corvo se a nova posição tiver uma solução melhor.

3. **Conclusão**
   - O algoritmo termina quando atinge o número máximo de iterações.

3.1.4. Modelo Matemático

O Binary Crow Search é um algoritmo metaheurístico inspirado no comportamento inteligente dos corvos. Nesta seção, apresentaremos a inspiração por trás do CSA e seu modelo matemático adaptado para problemas de otimização binária (BCS).

Os corvos são conhecidos pela sua inteligência e habilidades de memória. Eles vivem em bandos e podem lembrar-se dos locais onde esconderam o alimento. Além disso, os corvos, frequentemente, seguem outros bandos para roubar os esconderijos. Essas características fundamentais do comportamento dos corvos servem como base para a definição do CSA:

1. Os Corvos vivem em bandos.
2. Os Corvos lembram-se dos esconderijos.
3. Cada corvo segue outros bandos de corvos para roubar o alimento.
4. Os Corvos protegem os seus esconderijos de serem roubados por uma dada probabilidade.

3.1.4.1. Implementação do CSA

Definições e Parâmetros

- **N**: Número de corvos.
- **D**: Total de dimensões do problema.
- **AP**: Probabilidade de percepção (*Awareness Probability*).
- **FL**: Comprimento do voo (*Flight Length*).
- **tmax**: Número máximo de iterações.

Em cada iteração, $x_i^{(t)}$ representa a posição do corvo i na iteração t, onde $i = 1, 2, \ldots, N$ e $t = 1, 2, \ldots, tmax$. m_i é a melhor posição que o corvo i alcançou até ao momento (que é o seu esconderijo).

Atualização das Posições dos Corvos

Os corvos atualizam as sua posição de duas formas, determinadas pela probabilidade AP:

Caso 1: Corvo iii segue corvo jjj de forma despercebida:

$$x_i^{(t+1)} = x_i^{(t)} + FL \cdot (m_j^{(t)} - x_i^{(t)})$$

Caso 2: Corvo jjj percebe que está a ser perseguido:

$$x_i^{(t+1)} = x_i^{(t)} + r \cdot (2 \cdot u - 1)$$

onde r é um número aleatório uniformemente distribuído entre 0 e 1, e u é um vetor aleatório de bits 0 ou 1.

3.1.4.2. Implementação do BCS

Inicialização

A primeira etapa do BCS usa um processo de Bernoulli para gerar números binários aleatórios. Um número aleatório é gerado entre 0 e 1, e binarizado para 0 (se for menor que 0.5) ou 1 (se for maior ou igual a 0.5). Assim, completamos a inicialização para cada uma das variáveis D.

Operadores Booleanos:

O BCS utiliza operadores booleanos para atualizar as posições:

1. ***XOR* (^)**: Operador de diferença binária, eficaz para a otimização binária devido à sua probabilidade de mudança de 50%.
2. ***AND* (&)**: Operador usado em conjunto com *XOR* para gerar novas soluções vizinhas.

Atualização da Posição no BCS:

A posição de um corvo i é atualizada usando as memórias dos corvos i e j:

$$x_i^{(t+1)} = m_i^{(t)} \oplus (m_j^{(t)} \,\&\, r_b)$$

onde r_b é um número binário aleatório igual a 0 ou 1. A tabela de verdade (Tabela 2) mostra todas as combinações possíveis desse mecanismo.

Exploração

O parâmetro AP estabelece o equilíbrio entre exploração:

- Quando AP se aproxima de 0, o BCS foca na exploração local.
- Quando AP se aproxima de 1, o BCS realiza a exploração global.

Pseudo-código do BCS

1. Inicia a população de corvos aleatoriamente.
2. Avalia a aptidão de cada corvo.
3. Enquanto não atingir o critério de paragem (número máximo de iterações):
4. Para cada corvo i:
    1. Gera um número aleatório r.
    2. Se $r \geq AP$:
        - Atualiza a posição de i seguindo j.
    3. Caso contrário:
        - Atualiza a posição de i considerando a percepção de j.
    4. Avalia a nova posição.
    5. Atualiza a memória do corvo i se a nova posição for melhor.
5. Retorna a melhor solução encontrada.

Em suma…

Com o uso adequado de APe operações bitwise, o BCS é capaz de explorar eficientemente o espaço de procura para alcançar soluções ótimas ou quase ótimas.

3.1.5. Desempenho e Complexidade

O desempenho e a complexidade do Binary Crow Search são aspectos cruciais para avaliar a eficácia do algoritmo a resolver problemas de otimização, como o UFLP.
Nesta seção, exploramos a eficiência computacional do BCS e a sua capacidade de encontrar soluções de alta qualidade num tempo razoável.

3.1.5.1. Desempenho do BCS

O desempenho do BCS pode ser avaliado em termos de:

1. **Qualidade das Soluções**: A capacidade do BCS de encontrar soluções próximas ao ótimo ou mesmo ótimas para o problema em questão;

2. **Velocidade de Convergência**: A rapidez com que o algoritmo converge para a solução ótima;
3. **Estabilidade e Robustez**: A consistência do BCS em encontrar boas soluções em múltiplas execuções.

Para medir estes aspetos, são realizadas várias execuções do BCS e os resultados são comparados com outros algoritmos de otimização, tais como algoritmos genéticos, enxame de partículas, entre outros. Tipicamente, métricas como a média da função objetivo e o desvio padrão das soluções são usadas para avaliar o desempenho.

3.1.5.2. Complexidade Computacional

A complexidade computacional do BSC pode ser dividida em dois componentes principais:

- inicialização;
- iteração.

Inicialização

A complexidade da fase de inicialização do BCS é $O(N \cdot D)$, onde:

- N é o número de corvos (soluções).
- D é o número de dimensões (variáveis) do problema.

Durante a inicialização, cada corvo é posicionado aleatoriamente no espaço de pesquisa binário, e a memória de cada corvo é inicializada com sua posição inicial.

Iteração

A complexidade de cada iteração do BCS é $O(N \cdot D)$. Cada iteração envolve:

- Gerar uma nova posição candidata para cada corvo.
- Avaliar a função objetivo para cada nova posição.
- Atualizar a posição e a memória do corvo se a nova posição for melhor.

Assim, para um número máximo de iterações $tmax$, a complexidade total do BCS é $O(tmax \cdot N \cdot D)$.

3.1.5.3. Análise de Complexidade

A complexidade $O(tmax \cdot N \cdot D)$ do BCS indica que o tempo de execução do algoritmo cresce linearmente com o número de iterações, o número de corvos e o número de

dimensões. Este crescimento linear é favorável em comparação com outros algoritmos que podem ter complexidade exponencial em relação a um ou mais parâmetros.

Além disso, a simplicidade das operações bitwise *(XOR e AND)* contribui para a eficiência do algoritmo, permitindo atualizações rápidas das posições dos corvos.

3.1.6. Vantagens & Desvantagens

3.1.6.1. Vantagens:

- **Eficiência Computacional**: Devido à sua complexidade linear e às operações bitwise simples, o BCS é eficiente e rápido.
- **Equilíbrio entre Exploração e Exploração**: O uso do parâmetro AP permite um bom equilíbrio entre a exploração de novas áreas e a exploração de áreas conhecidas.
- **Facilidade de Implementação**: O BCS é relativamente fácil de implementar, com poucas etapas e operações simples.

3.1.6.1. Desvantagens:

- **Dependência de Parâmetros**: O desempenho do BSC pode ser sensível à escolha dos parâmetros, como o número de corvos, a probabilidade de percepção AP e o número máximo de iterações $tmax$.
- **Possível Convergência Prematura**: Tal como muitos algoritmos metaheurísticos, o BSC pode, em alguns casos, convergir prematuramente para soluções subótimas, especialmente se o equilíbrio entre exploração e exploração não for bem ajustado.

3.1.7. Aplicação no mundo Real

O Binary Crow Search tem sido aplicado em diversos cenários reais, o que demonstra a sua eficácia e versatilidade. A aplicação no UFLP é especialmente notável.

3.1.7.1. Aplicação no UFLP

No contexto do UFLP, o BCS pode ser utilizado para explorar o espaço de pesquisa binária e encontrar a configuração ótima das instalações. A abordagem do BCS envolve a inicialização de uma população de soluções (corvos), onde cada corvo representa uma potencial configuração de instalações. O BCS equilibra a exploração de novas áreas e a exploração de áreas já conhecidas para encontrar soluções ótimas ou quase ótimas de forma eficiente.

Exemplo de Aplicação:

No caso de uma empresa que deseja abrir uma série de armazéns para distribuir produtos aos seus clientes. O BCS poderia ser utilizado para determinar os melhores locais para esses armazéns, considerando fatores como os custos de transporte, a proximidade dos clientes e a capacidade dos armazéns. A empresa poderia utilizar o BCS para explorar várias configurações possíveis e encontrar a solução que minimize os custos totais da operação.

3.1.7.2. Outras Aplicações Reais

Além do UFLP, o BCS pode ser aplicado numa variedade de problemas de otimização binária, incluindo:

- **Planeamento de Redes de Comunicação:** Determinação dos melhores locais para instalar antenas de telemóvel para maximizar a cobertura e minimizar os custos associados.
- **Design de Circuitos Eletrónicos:** Otimização do layout de componentes eletrónicos para minimizar interferências e maximizar a eficiência.
- **Gestão de Recursos Energéticos:** Determinação das melhores estratégias para a distribuição de energia elétrica para reduzir perdas e melhorar a eficiência.

3.1.8. Análise de Resultados

Instance	Mean	median	Min (best)	Max (worst)	STD Deviation	Number of Comparisons	Optimal Solution
capa	*4.535.44e+07*	*4.31539e+07*	*1.171565e+07*	*8.31141e+07<*	*2.05819e+07*	*1858367171*	*17156454.478*
capb	*2.24455e+07*	*1.96563e+07*	*1.29791e+07*	*4.47827e+07*	*9.10015e+06*	*4034047065*	*12979071.582*
capc	*1.80457e+07*	*1.63003e+07*	*1.15094e+07*	*3.33083e+07*	*6.27815e+06*	*5591143588*	*11505594.329*
mr1	*32285.9*	*26823.1*	*2349.86*	*89036.6*	*24541*	*11846773618*	*2349.856*
mr2	*31210.8*	*27211.1*	*2344.76*	*76475.9*	*20054.8*	*11487571098*	*2344.757*
mr3	*34274.2*	*36545.2*	*2183.23*	*81194.7*	*22343.5*	*12151243068*	*2183.235*
mr4	*36014.3*	*41858.8*	*2433.11*	*95719.7*	*26815.8*	*12175369344*	*2433.110*
mr5	*42209.7*	*41858.8*	*2344.35*	*84928.9*	*25576.5*	*11810625422*	*2344.353*

Tabela 1: Resultados do Algoritmo BCS em diferentes Instâncias do UFLP

A tabela 1 apresenta os resultados do algoritmo BCS aplicados a diversas instâncias do UFLP. As métricas avaliadas incluem a média, mediana, valor mínimo (melhor), valor máximo (pior) e o desvio padrão das soluções obtidas para cada instância. As instâncias são rotuladas como *capa, capb, capc, mr1, mr2, mr3, mr4* e *mr5*.

Instâncias capa, capb e capc

1. ***capa***

   A instância *capa* apresenta a maior média entre todas as instâncias, indicando que, em geral, as soluções encontradas são mais caras. O desvio padrão relativamente alto sugere uma grande variabilidade nas soluções, o que pode indicar que o algoritmo explorou uma ampla gama de possíveis soluções.

2. ***capb***

   A instância *capb* tem uma média menor que *capa*, indicando soluções geralmente mais económicas. O desvio padrão é também menor que *capa*, mas ainda significativo, apontando para uma variação notável nas soluções.

3. ***capc***

   A instância *capc* apresenta a menor média e desvio padrão entre as instâncias de *capa*, sugerindo soluções mais estáveis e geralmente mais económicas.

Instâncias mr1, mr2, mr3, mr4 e mr5

1. ***mr1***
   A instância *mr1* mostra uma alta variabilidade com um desvio padrão significativo. A diferença entre o valor mínimo e máximo é grande, indicando que o algoritmo encontrou tanto soluções muito boas quanto soluções relativamente fracas.
2. ***mr2***
   A instância *mr2* tem resultados semelhantes ao *mr1*, mas com uma menor variabilidade, sugerindo soluções mais consistentes.
3. ***mr3***
   A instância *mr3* apresenta a maior média entre as instâncias de *mr*, indicando soluções geralmente mais caras. O desvio padrão também é elevado, refletindo uma alta variabilidade nas soluções.

4. ***mr4***

   A instância *mr4* tem a maior variabilidade, com o maior desvio padrão, indicando uma ampla gama de soluções encontradas pelo algoritmo.

5. ***mr5***

   A instância *mr5* possui a maior média, sugerindo soluções geralmente mais dispendiosas. O desvio padrão também é alto, indicando variabilidade significativa nas soluções.

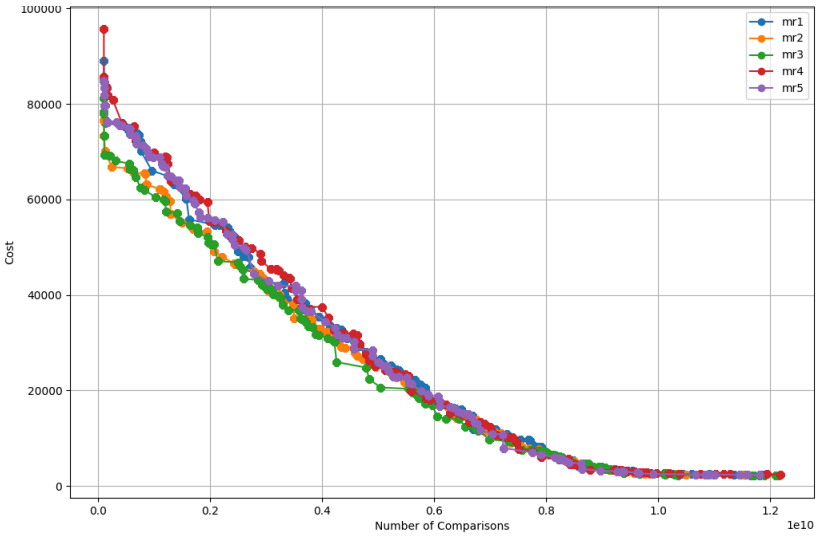


Figura 1: Number of Comparisons & Cost Evolution by MR Problem

A Figura 1 mostra a evolução dos custos em função do número de comparações para as instâncias mr (mr1 a mr5). Todas as instâncias apresentam uma tendência decrescente no custo, indicando que o algoritmo BCS refina bem as soluções. Após um certo número de comparações, todas convergem para custos semelhantes, mostrando a estabilidade do algoritmo. Inicialmente, há maior variabilidade nos custos devido às características específicas de cada instância.

Cada instância tem um comportamento inicial diferente, mas todas convergem para custos baixos de forma consistente, demonstrando a eficácia e a consistência do algoritmo.

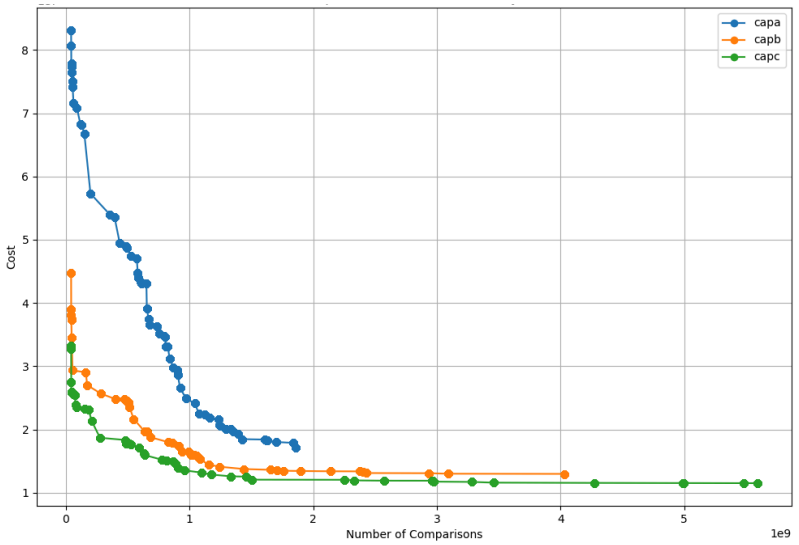
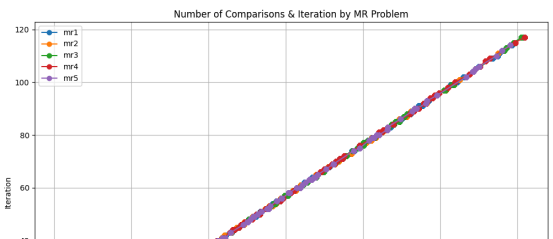


Figura 2: Number of Comparisons & Cost Evolution by CAP Problem

A Figura 2 apresenta a evolução dos custos em função do número de comparações para as instâncias CAP (capa, capb, capc). Cada linha colorida representa uma instância diferente do problema CAP. Todas as instâncias cap mostram uma tendência decrescente no custo à medida que o número de comparações aumenta, indicando que o algoritmo BCS está bem implementado, refinando as soluções e encontrando custos menores com mais iterações. Observa-se que todas as instâncias convergem para custos próximos de 1, mostrando a estabilidade do algoritmo em encontrar soluções ótimas ou próximas do ótimo. Nos estágios iniciais (número de comparações abaixo de 1 x 10^9), há uma maior variabilidade nos custos entre as diferentes instâncias, devido às características específicas de cada instância e à fase exploratória inicial do algoritmo. Especificamente, a instância capa (linha azul) começa com um custo acima de 8.000.000 e rapidamente cai para cerca de 4.000.000, seguindo depois uma descida gradual até se estabilizar próximo de 1.000.000, indicando uma eficiência razoável do algoritmo. A instância capb (linha laranja) apresenta um comportamento semelhante ao capa, começando em torno de 4.000.000 e descendo de forma consistente, com uma convergência final bem alinhada com as outras instâncias, mostrando boa eficácia do algoritmo. A instância capc (linha verde) começa ligeiramente abaixo de 4.000.000 e tem uma descida mais suave e consistente, alcançando soluções de custo mais baixo relativamente rápido, mostrando boa eficácia do algoritmo nesta instância.



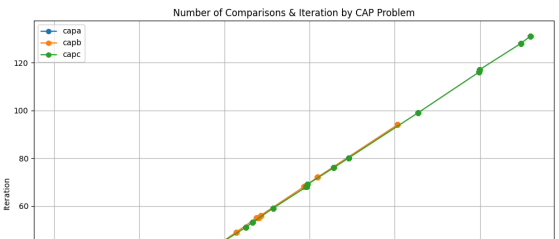


Figura 3 & 4: Number of Comparisons & Iteration by MR & CAP

As figuras apresentadas mostram o número de comparações e iterações para dois problemas diferentes:

1. **Problema MR**:
    - O gráfico à esquerda exibe uma correlação linear quase perfeita entre o número de comparações e o número de iterações para várias instâncias do problema MR (mr1 a mr5).
    - Isso indica que o número de comparações aumenta consistentemente com o número de iterações, sugerindo uma complexidade constante e elevada em todas as iterações.
2. **Problema CAP**:
    - Inicialmente, o número de comparações é elevado, mas à medida que o número de iterações aumenta, o número de comparações começa a diminuir, mostrando uma tendência de otimização e redução das comparações ao longo do tempo.

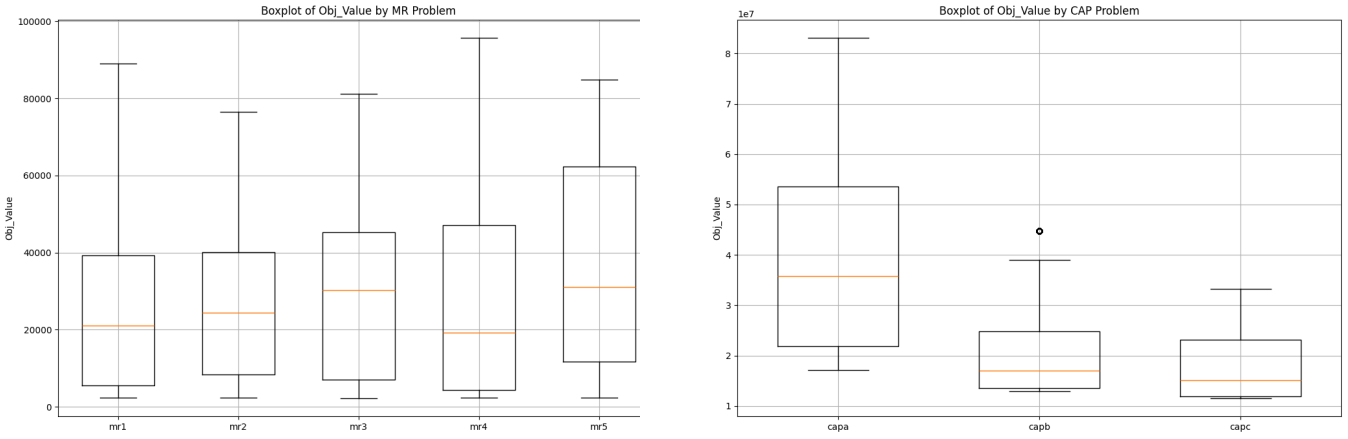
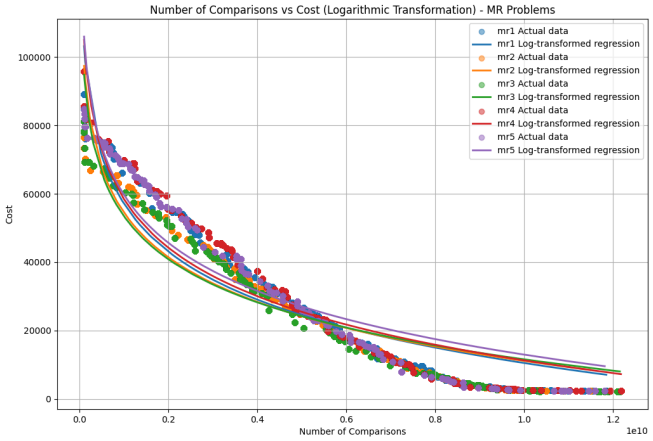


Figura 5 & 6: Boxplot of Objective value by MR & CAP

1. **Boxplot do Problema “MR”**:
    - Os valores dos objetivos (Obj_Value) variam entre aproximadamente 0 e 100,000.

- A mediana dos valores dos objetivos está em torno de 20,000 para todos os problemas “MR” (mr1 a mr5).
- A distribuição dos valores dos objetivos é semelhante entre os diferentes problemas “MR”, com alguma variação na dispersão e nos valores extremos.

2. **Boxplot do Problema “CAP”**:
   - Os valores dos objetivos variam entre aproximadamente 1 milhão e 8 milhões.
   - A mediana dos valores dos objetivos é maior no problema “capa” comparado aos problemas “capb” e “capc”.
   - O problema capa tem a maior variação nos valores dos objetivos, enquanto “capb” tem a menor variação, com um outlier significativo.



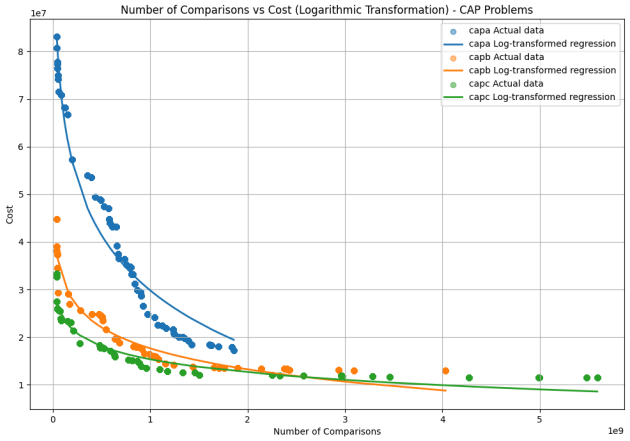


Figura 7 & 8: Number of Comparisons bs Cost for MR & CAP

As figuras apresentam gráficos de dispersão com regressões logarítmicas que mostram a relação entre o número de comparações e o custo (cost) para os problemas “MR” e “CAP”.

Problemas MR:

- O custo diminui com o aumento do número de comparações.
- As regressões logarítmicas ajustam-se bem aos dados, mostrando uma tendência decrescente consistente para todos os problemas” MR” (“mr1” a “mr5”).

Problemas CAP:

- O custo também diminui com o aumento do número de comparações.

- As regressões logarítmicas mostram uma tendência decrescente para todos os problemas “CAP” (“capa”, “capb”, “capc”), embora a variação nos custos seja maior do que nos problemas “MR”.

Conclusões Gerais da Análise de Resultados

A análise dos resultados do Algoritmo BCS em diferentes instâncias do Problema de Localização de Instalações foi realizada com base em três figuras distintas, focando nas instâncias *CAP* e *MR*.

O Algoritmo BCS demonstrou ser uma abordagem robusta e eficiente para resolver problemas de localização de instalações com restrições de capacidade (*CAP*) e sem restrições (*MR*). A análise dos resultados revelou:

- **Eficácia na Redução de Custos:** O algoritmo conseguiu reduzir significativamente os custos em todas as instâncias, tanto *CAP* quanto *MR*, especialmente nas fases iniciais de execução.
- **Consistência e Convergência:** Apesar das variações iniciais nos resultados, o algoritmo mostrou consistência ao convergir para soluções de baixo custo em todas as instâncias analisadas.
- **Robustez:** A capacidade do Algoritmo BCS de lidar com diferentes características de problemas e instâncias, alcançando soluções competitivas e estáveis, destaca a sua robustez e eficiência.

Em suma, o Algoritmo BCS é uma ferramenta eficaz para resolver UFLP, apresentando bom desempenho em diferentes cenários e mostrando-se capaz de encontrar soluções de alta qualidade de forma consistente e eficiente.

3.2 Algoritmo Simulated Annealing

3.2.1 História

O Simulated Annealing tem como inspiração uma técnica física conhecida como recozimento, um processo utilizado na metalurgia para aumentar a qualidade dos materiais. Em 1983, S. Kirkpatrick, C.D. Gelatt e M.P. Vecchi introduziram o conceito no campo da otimização combinatória no seu artigo seminal, adaptando os princípios do recozimento para resolver problemas complexos de otimização. Eles mostraram que, assim como o recozimento físico permite que os átomos de um metal encontrem uma estrutura de energia mínima, o SA pode guiar uma solução para um mínimo global de uma função de custo.

Nos anos seguintes, o SA foi rapidamente adotado em diversas áreas da engenharia e ciência. Durante a década de 1980 e início da década de 1990, o algoritmo foi aplicado a problemas de design de circuitos, planeamento de rotas, layout de fábricas, e muito mais. A simplicidade e a flexibilidade do SA, em conjunto com a capacidade de escapar de mínimos locais, faziam deste algoritmo uma ferramenta valiosa em situações onde métodos exatos eram impraticáveis devido à alta complexidade computacional.

A evolução do SA ao longo dos anos foi marcada pela introdução de diversas melhorias e variações. Investigadores experimentaram diferentes esquemas de arrefecimento,

métodos de formação de vizinhos e técnicas de aceitação de soluções para aumentar a eficiência e a robustez do algoritmo. Um avanço significativo nesta área foi a introdução de técnicas híbridas, combinando o SA com outros métodos heurísticos, como a *Tabu Search* e algoritmos genéticos, para aproveitar os pontos fortes das múltiplas abordagens.

Na década de 2000, o SA continuou a evoluir com a incorporação de técnicas de *Machine Learning* e inteligência artificial. Essas inovações permitiram ajustes automáticos dos parâmetros do algoritmo e a adaptação dinâmica durante o processo de otimização, melhorando ainda mais a sua performance e aplicabilidade.

Olhando para o futuro, o SA permanece num campo ativo de pesquisa. Com o advento da computação quântica, há um interesse crescente em adaptar os princípios do SA para aproveitar o potencial das novas tecnologias quânticas, prometendo soluções ainda mais rápidas e eficazes para problemas de otimização combinatória. Além disso, a integração do SA com big data e análise preditiva abre novas fronteiras para aplicações em tempo real e em larga escala, mantendo o SA como uma ferramenta relevante e poderosa para resolver problemas complexos em diversas disciplinas.

3.2.2 Introdução do método relativamente ao UFLP

Devido à complexidade combinatória do UFLP, métodos exatos tornam-se impraticáveis para grandes instâncias, fazendo com que heurísticas como o Simulated Annealing sejam uma escolha natural.

O Simulated Annealing é particularmente adequado para o UFLP devido à sua capacidade de escapar de mínimos locais e explorar um espaço de soluções mais amplo. Esta técnica permite aceitar soluções temporariamente piores para potencialmente alcançar soluções melhores a longo prazo, o que é essencial para problemas onde a paisagem de custo é complexa e cheia de picos e vales.

3.2.3 Descrição do Algoritmo

O Simulated Annealing (SA) é uma heurística inspirada no processo físico de recozimento, utilizado para resolver problemas complexos de otimização como o Problema de Localização de Instalações sem Restrições de Capacidade (UFLP). A técnica envolve a formação e avaliação de soluções vizinhas, aceitando ocasionalmente soluções piores para escapar de mínimos locais, com a probabilidade de aceitação decrescendo ao longo do tempo conforme a temperatura diminui.

3.2.3.1 Passo a Passo

1. **Inicialização**
    - Definir os parâmetros do algoritmo
    - Gerar uma solução inicial aleatória
        - Atribuir cada cliente a um armazém aleatoriamente
    - Calcular o custo total da solução inicial
2. **Iteração**
    - Enquanto a temperatura atual for maior que a temperatura final, para um número definido de iterações por temperatura
        - Gerar uma solução vizinha perturbando a solução atual
            i. Realocar aleatoriamente um subconjunto de clientes a diferentes armazéns.
        - Aplicar uma *Local Search* para refinar a solução vizinha
            i. Utilizar uma lista tabu para evitar ciclos e melhorar a solução.
    - Avaliar e aceitar a nova solução
        - Calcular a diferença de custo entre a solução atual e a nova solução
        - Se a nova solução tiver um custo menor, aceitá-la
        - Se a nova solução tiver um custo maior, aceitá-la com uma probabilidade que diminui exponencialmente com o aumento do custo e a diminuição da temperatura
        - Atualizar a melhor solução encontrada se a solução atual for melhor
    - Reduzir a temperatura.
    - Aplicar uma perturbação adaptativa periodicamente para escapar de mínimos locais
3. **Conclusão**
    - O algoritmo termina quando a temperatura atinge um valor final predeterminado ou após o número máximo de iterações
    - Compilar e retornar a solução final.

3.2.4 Modelo Matemático

O modelo matemático do Simulated Annealing baseia-se em conceitos probabilísticos e físicos, adaptados para resolver problemas de otimização combinatória. No contexto do UFLP, o modelo matemático do SA pode ser descrito da seguinte forma:

3.2.4.1. Função Objetivo:

O objetivo do UFLP é minimizar o custo total, que inclui os custos fixos de abertura dos armazéns e os custos variáveis de transporte dos clientes para os armazéns. A função objetivo pode ser formulada como:

$$C_{total} = \sum_{j \in J} c_{ij} \; + \; \sum_{i \in I} f_i$$

onde:

- c_{ij} é o custo de alocação do cliente j ao armazém i,
- f_i é o custo fixo de abertura do armazém i,
- I é o conjunto de armazéns,
- J é o conjunto de clientes.

3.2.4.2. Perturbação e Formação de Vizinhos:

Para explorar o espaço de soluções, o SA gera uma nova solução S' a partir da solução atual S através de pequenas perturbações. No UFLP, isso envolve realocar aleatoriamente um subconjunto de clientes a diferentes armazéns. A nova solução S' é avaliada e comparada com a solução atual S.

3.2.4.3. Critério de Aceitação

O critério de aceitação é fundamental para o funcionamento do SA, permitindo que sejam aceites soluções piores com uma certa probabilidade para escapar de mínimos locais. A taxa de aceitação P de uma solução pior é dada por:

$$P \; = \; exp\left(\frac{-\Delta c}{T}\right)$$

onde:

- $\Delta C \; = \; C_{total}(S') \; - \; C_{total}(S)$ é a diferença de custo entre a nova solução S' e a solução atual S;
- T é a temperatura atual, um parâmetro que controla a taxa de aceitação de soluções piores.

3.2.4.4. Atualização da Temperatura

A temperatura T é gradualmente reduzida ao longo das iterações de acordo com um esquema de arrefecimento predefinido. Um esquema comum é a redução exponencial:

$$T_{nova} = aT_{atual}$$

onde: $0 < a < 1$ é a taxa de arrefecimento

3.2.4.5. Iteração e Convergência

O SA itera sobre o processo de formação de vizinhos, avaliação e aceitação de soluções e atualização da temperatura até que um critério de paragem seja satisfeito (tal como atingir uma temperatura final T_{final} ou completar um número máximo de iterações.

3.2.4.6. Modelo Passo a Passo

1. **Inicialização**
   - ➔ Definir $T_{inicial}$, T_{final}, a, e número máximo de iterações
   - ➔ S = solução inicial aleatória
   - ➔ $C_{total}(S) = \sum_{j \in J} c_{ij} + \sum_{i \in I} f_i$
2. **Iteração**
   - ➔ Enquanto $T > T_{final}$ e número de iterações não for atingido:
     - ◆ S' = gerar solução vizinha (S)
     - ◆ $\Delta C = C_{total}(S') - C_{total}(S)$
     - ◆ Se $\Delta C < 0$ ou $exp\left(\frac{-\Delta c}{T}\right) > rand()$ então
     - ◆ $S = S'$
   - ➔ Atualizar a melhor solução encontrada se $C_{total}(S)$ for menor
3. **Conclusão**
   - ➔ Retornar a melhor solução encontrada.

3.2.5 Desempenho e Complexidade

O desempenho do Simulated Annealing é amplamente influenciado pela configuração dos parâmetros, incluindo a temperatura inicial, a taxa de arrefecimento, o número de iterações por temperatura e o critério de paragem. Estes parâmetros determinam a capacidade do algoritmo de explorar o espaço de soluções e fugir dos mínimos locais.

3.2.5.1. Desempenho

O SA é eficaz na obtenção de soluções de alta qualidade para problemas de otimização complexos como o UFLP. A sua capacidade de aceitar soluções piores temporariamente, permite uma exploração mais ampla do espaço de soluções, o que é crucial para evitar a estagnação em mínimos locais. A eficiência do SA pode ser melhorada através da aplicação de técnicas como Local Search e perturbações adaptativas.

A performance do SA pode ser avaliada em termos de tempo de execução e qualidade das soluções encontradas. Embora o SA não garanta a obtenção da solução ótima global, ele tende a encontrar soluções próximas ao ótimo dentro de um tempo computacionalmente viável. A qualidade da solução melhora com a diminuição gradual da temperatura, permitindo uma convergência mais fina na fase final do algoritmo.

3.2.5.2. Complexidade

A complexidade computacional do SA para o UFLP pode ser expressa como $O(N \cdot I)$, onde N é o número de clientes e I é o número de iterações. Cada iteração envolve o desenvolvimento de uma solução vizinha, a avaliação do custo da nova solução e a decisão de aceitação (baseada no critério probabilístico). Adicionalmente, a aplicação do Local Search adiciona uma camada extra de complexidade, pois envolve a exploração detalhada das soluções vizinhas.

O uso de uma lista tabu na pesquisa local evita ciclos e melhora a eficiência, embora, por outro lado, contribua para o aumento da complexidade computacional. A implementação eficiente da lista tabu é crucial para manter o balanceamento entre exploração e exploração.

3.2.5.3. Considerações Práticas

Para melhorar o desempenho e gerir a complexidade, várias estratégias podem ser adotadas, tais como:

- **Afinamento dos Parâmetros:** Ligeiros ajustes na temperatura inicial, taxa de arrefecimento e número de iterações podem otimizar o balanceamento entre exploração e convergência.
- **Heurísticas Híbridas:** A combinação do SA com outras heurísticas, como algoritmos genéticos ou *Tabu Search*, pode aproveitar as vantagens de múltiplas abordagens, melhorando a robustez e a eficiência do algoritmo.
- **Paralelização:** A execução paralela de múltiplas instâncias do SA ou a paralelização das avaliações de custos podem reduzir significativamente o tempo de execução.

3.2.6 Vantagens & Desvantagens

3.2.6.1. Vantagens

1. **Capacidade de Escapar de Mínimos Locais:** Uma das maiores forças do SA é a sua habilidade em aceitar temporariamente soluções piores, permitindo que o algoritmo fuja de mínimos locais e explore regiões do espaço de soluções que poderiam ser inacessíveis para métodos mais rígidos. Isso é crucial para problemas como o UFLP, onde a paisagem de custo pode ser complexa e repleta de armadilhas locais.
2. **Flexibilidade e Simplicidade de Implementação:** O SA é relativamente simples de implementar, não exigindo derivadas ou gradientes da função objetivo, o que o torna aplicável a uma vasta gama de problemas. A sua simplicidade estrutural facilita a adaptação e integração com outras técnicas heurísticas e meta-heurísticas.
3. **Adequação para Problemas Combinatórios Grandes:** O SA é particularmente eficaz em lidar com problemas de grande escala, onde métodos exatos são impraticáveis devido à explosão combinatória. Para o UFLP, essa característica é vital, já que as instâncias grandes são comuns e a solução exata pode ser computacionalmente inviável.
4. **Aplicabilidade Geral:** Devido à sua natureza probabilística e flexível, o SA pode ser aplicado a diversos tipos de problemas de otimização em diferentes domínios, como planeamento logístico, design de circuitos, e muitos outros, o que o torna valioso e versátil.
5. **Robustez a Modificações do Problema:** O SA pode ser facilmente adaptado para lidar com variações e modificações no problema original sem necessitar de uma reformulação completa do algoritmo. Isto é útil em contextos dinâmicos onde os requisitos e restrições do problema podem mudar ao longo do tempo.

3.2.6.2. Desvantagens

1. **Sensibilidade à Escolha dos Parâmetros:** A performance do SA é altamente dependente da escolha apropriada dos parâmetros, como a temperatura inicial, a taxa de arrefecimento e o número de iterações. A escolha inadequada desses parâmetros pode levar a uma exploração insuficiente ou excessiva do espaço de soluções, resultando numa performance subótima.
2. **Não é Garantido o Alcance da Solução Ótima Global:** Embora o SA seja eficaz em encontrar soluções de alta qualidade, ele não é capaz de garantir a obtenção da solução ótima global. O algoritmo pode convergir para uma solução subótima, especialmente se a taxa de arrefecimento for muito rápida ou se não houver iterações suficientes para uma exploração adequada.
3. **Intensidade Computacional:** O SA pode ser computacionalmente intensivo, especialmente para grandes instâncias de problemas como o UFLP. A necessidade de calcular repetidamente os custos das soluções vizinhas e a aplicação de técnicas adicionais, como a *Local Search*, aumentam a carga computacional.
4. **Dependência do Tempo de Execução:** O tempo de execução do SA pode ser significativamente longo, especialmente para alcançar soluções de alta qualidade. A convergência lenta pode ser uma limitação em contextos onde a solução rápida é crítica.
5. **Possibilidade de Convergência Prematura:** O SA pode, em alguns casos, convergir prematuramente para uma solução subótima, especialmente se a taxa de aceitação de soluções piores diminuir muito rapidamente. Isso pode ocorrer se a temperatura for arrefecida de forma muito exponencial.
6. **Necessidade de Ajuste Sensível dos Parâmetros:** O ajuste sensível dos parâmetros do SA requer tentativa e validação, o que pode ser um processo demorado e exigente. Sem um ajuste cuidadoso, o algoritmo pode não alcançar o seu potencial máximo de performance.
7. **Implementação de Técnicas Complementares:** Embora a aplicação de técnicas complementares, como o *Local Search* e perturbações adaptativas, possam melhorar a performance do SA, essas adições aumentam a complexidade da implementação e podem exigir conhecimento especializado para serem configuradas corretamente.

Em suma, o Simulated Annealing é uma heurística poderosa e flexível com diversas vantagens que a tornam uma escolha atraente para resolver problemas complexos de otimização. No entanto, as desvantagens, principalmente relacionadas à sensibilidade dos parâmetros e à intensidade computacional, necessitam especial atenção de forma a maximizar a eficácia do algoritmo.

3.2.7 Aplicação no mundo Real

O Simulated Annealing tem sido amplamente aplicado em diversos campos devido à sua capacidade de encontrar soluções aproximadas de alta qualidade para problemas complexos de otimização. Aqui estão algumas das áreas onde o SA tem sido especialmente útil:

1. **Logística e Gestão:**
    - **Localização de Armazéns:** O SA é usado para determinar a localização ideal de armazéns e centros de distribuição, minimizando os custos totais de operação e transporte.
    - **Roteirização de Veículos:** O algoritmo ajuda a otimizar rotas de entrega para veículos, reduzindo o tempo de viagem e o consumo de combustível.
2. **Planeamento Urbano e Infraestruturas:**
    - **Planeamento de Redes de Transporte:** O SA é utilizado para otimizar a disposição de redes de transporte urbano, incluindo rotas de transportes públicos para melhorar a eficiência e a rede de cobertura.
    - **Gestão de Tráfego:** O algoritmo ajuda a otimizar a sinalização de trânsito e o controlo de semáforos, minimizando engarrafamentos e melhorando o fluxo de veículos.
3. **Design de Circuitos Eletrónicos:**
    - ***Layout* de Circuitos Integrados:** O SA é aplicado no design de circuitos VLSI (Very-Large-Scale Integration) para minimizar o comprimento das conexões e o atraso de sinal, resultando em circuitos mais rápidos e eficientes.
    - **Distribuição de Componentes:** O algoritmo ajuda a otimizar a colocação de componentes eletrónicos em placas de circuitos (PCB), reduzindo interferências e melhorando a performance.
4. **Engenharia de Software:**
    - **Testes de Software:** O SA é utilizado para gerar casos de teste que maximizam a cobertura de código e identificam possíveis falhas de forma eficiente.
    - ***Refactoring* de Código:** O algoritmo ajuda a identificar a melhor maneira de reorganizar o código para melhorar a legibilidade e manutenção, sem alterar a funcionalidade.
5. **Telecomunicações:**

- **Planeamento de Redes de Comunicação:** O SA é usado para otimizar a disposição de antenas de telemóvel e outros equipamentos de comunicação, garantindo cobertura máxima e minimizando interferências.
- **Alocação de Frequências:** O algoritmo ajuda a distribuir frequências de forma eficiente entre diferentes transmissores, evitando interferências e maximizando a utilização do espectro.

6. **Bioinformática:**
   - **Alinhamento de Sequências:** O SA é aplicado para encontrar o melhor alinhamento entre sequências de DNA, RNA ou proteínas, o que é crucial para a análise comparativa e a identificação de funções biológicas.
   - **Modelagem de Estruturas de Proteínas:** O algoritmo é usado para prever a estrutura tridimensional de proteínas com base na sua sequência de aminoácidos, ajudando na compreensão das suas funções e interações.
7. **Financeiro e Económico:**
   - **Portfólios de Investimentos:** O SA é utilizado para otimizar a composição de carteiras de investimento, equilibrando risco e retorno de acordo com as preferências dos investidores.
   - **Gestão de Recursos:** O algoritmo ajuda na alocação eficiente de recursos financeiros em diferentes projetos ou departamentos, maximizando o retorno sobre o investimento.
8. **Inteligência Artificial e *Machine Learning*:**
   - ***Neural Networks*:** O SA é utilizado para ajustar os pesos de redes neurais, encontrando combinações que minimizem o erro de previsão.
   - **Otimização de Hiperparâmetros:** O algoritmo ajuda a selecionar os melhores hiperparâmetros para modelos de machine learning, melhorando a sua performance.

Em resumo, o Simulated Annealing é uma ferramenta versátil e poderosa, aplicável a uma ampla variedade de problemas reais. A capacidade de fornecer soluções eficientes e de alta qualidade para problemas complexos faz dele um recurso valioso em muitos campos da ciência e da engenharia.

3.2.8 Análise de Resultados

Instance	Mean	median	Min (best)	Max (worst)	STD Deviation	Number of Comparisons	Optimal Solution

capa	*3.13586e+07*	*3.13606e+07*	*3.08303e+07*	*3.33607e+07*	*476821*	*15995837924*	*17156454.478*
capb	*1.73098e+07*	*1.6446e+07*	*1.57938e+07*	*2.0921e+07*	*1.54517e+06*	*8314905235*	*12979071.582*
capc	*1.88397e+07*	*1.86282e+07*	*1.6808e+07*	*2.2533e+07*	*1.76913e+06*	*21834127304*	*11505594.329*
mr1	*3872.24*	*3871.38*	*3766.27*	*4049.08*	*64.5237*	*51953452168*	*2349.856*
mr2	*3489.74*	*3449.69*	*3391.8*	*3776.57*	*99.4568*	*53715084323*	*2344.757*
mr3	*3757.44*	*3515.77*	*3416.61*	*4412.88*	*387.413*	*46113862449*	*2183.235*
mr4	*4484.25*	*4472.13*	*4347.95*	*4749.4*	*112.6*	*50605085180*	*2433.110*
mr5	3790.71	3724.05	3623.79	4160.18	187.145	15059086547	*2344.353*

Tabela 2: Resultados do Algoritmo SA em diferentes Instâncias do UFLP

A tabela 2 apresenta os resultados do algoritmo SA aplicados a diversas instâncias do UFLP. As métricas avaliadas incluem a média, mediana, valor mínimo (melhor), valor máximo (pior) e o desvio padrão das soluções obtidas para cada instância. As instâncias são rotuladas como *capa, capb, capc, mr1, mr2, mr3, mr4* e *mr5*.

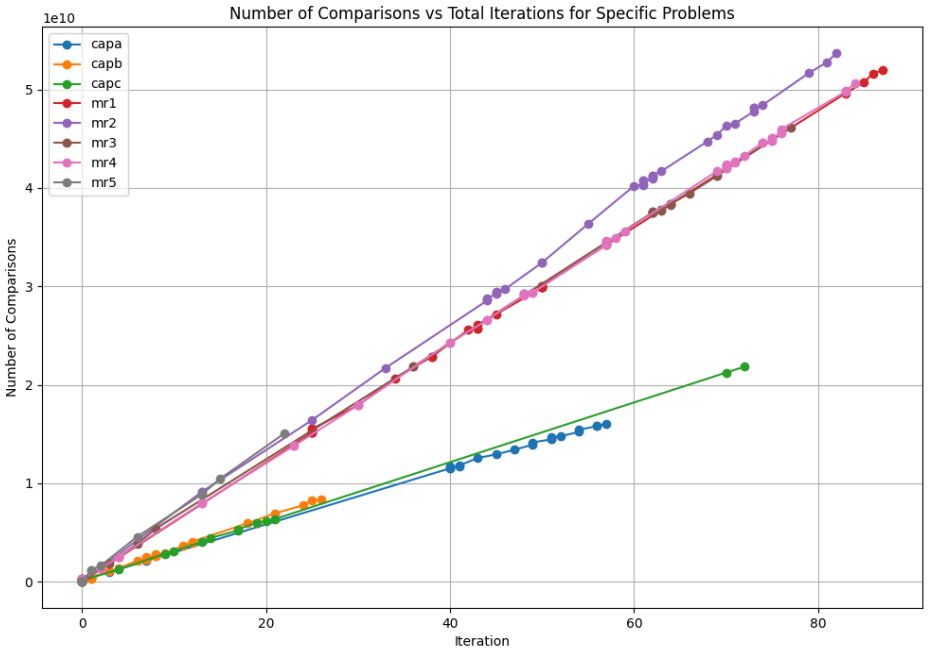


Figura 9: Number of Comparisons & Total Iterations by CAP & MR

Na figura 9, é possível observar um facto extremamente interessante acerca do comportamento do algoritmo: em instâncias com custos elevados, o problema consegue terminar em menos iterações e com menos comparações. Por outro lado, nos problemas do conjunto M*, é visível o desafio enfrentado pelo algoritmo ao lidar com numerosos valores semelhantes, o que demonstra uma maior complexidade.

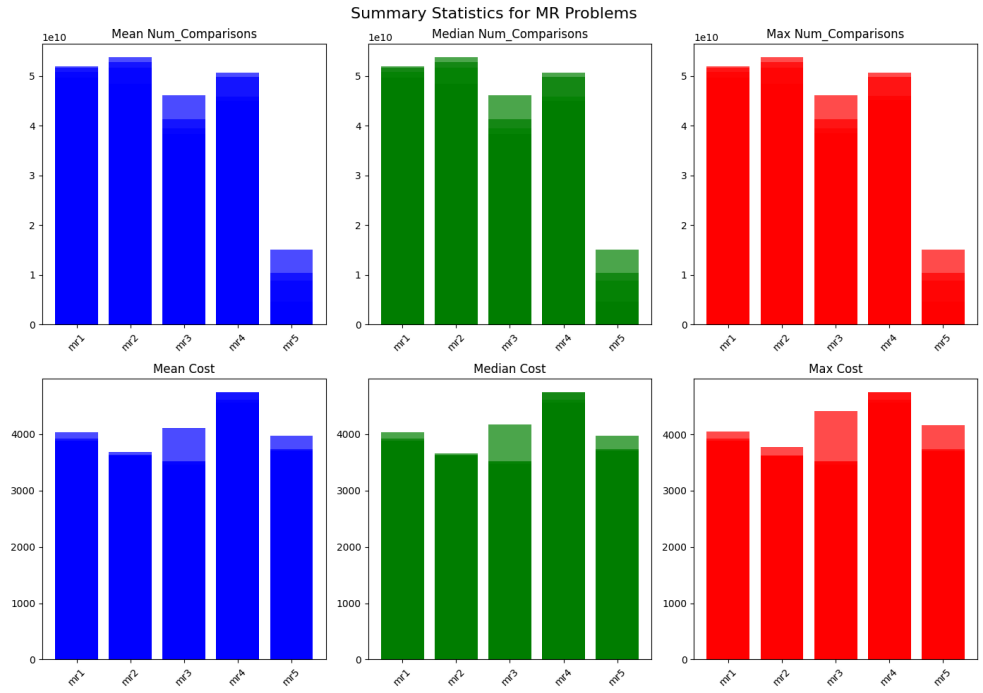


Figura 10: Summary of MR

Os gráficos fornecidos apresentam as estatísticas sumárias de um algoritmo *Simulated Annealing* aplicado a problemas MR, destacando três métricas principais: número de comparações, custo médio e custo máximo. Cada métrica é analisada em termos de média, mediana e valor máximo, comparando cinco variantes de problemas MR (mr1 a mr5).

Em contraste, a variante mr4 mostra uma redução significativa, sugerindo uma menor complexidade ou maior eficiência na convergência do algoritmo. Já a variante mr5 apresenta o menor número médio de comparações, indicando uma eficiência superior ou menor complexidade.

A mediana das comparações segue um padrão similar às médias, com mr1, mr2 e mr3 no topo, enquanto mr4 e mr5 mostram valores menores, reforçando a ideia de que estes problemas exigem menos comparações. Isso pode ser devido às características intrínsecas dos problemas ou à eficácia do algoritmo nestas variantes.

Quando analisamos o número máximo de comparações, os valores mais altos são observados em mr1, mr2 e mr3, enquanto mr4 e mr5 apresentam números significativamente menores. Esta variação sugere que as instâncias dos problemas mr1, mr2 e mr3 podem ser mais difíceis, necessitando de mais comparações.

Em relação ao custo, a análise mostra que o custo médio é maior em mr4, seguido de perto por mr1 e mr3. As variantes mr2 e mr5 têm custos médios menores, o que sugere que o algoritmo encontra soluções mais econômicas nestes casos. A mediana dos custos reflete a mesma tendência, com mr4 tendo a mediana mais alta, indicando uma tendência para soluções mais caras. A variante mr3 apresenta uma mediana superior à média, possivelmente devido à presença de algumas instâncias com soluções significativamente mais caras.

O custo máximo acompanha o padrão observado nos custos médios e medianos, com mr4 apresentando os valores mais altos, seguido por mr3 e mr1. As variantes mr2 e mr5 continuam a ter os custos máximos mais baixos, indicando que essas instâncias tendem a ser menos onerosas.

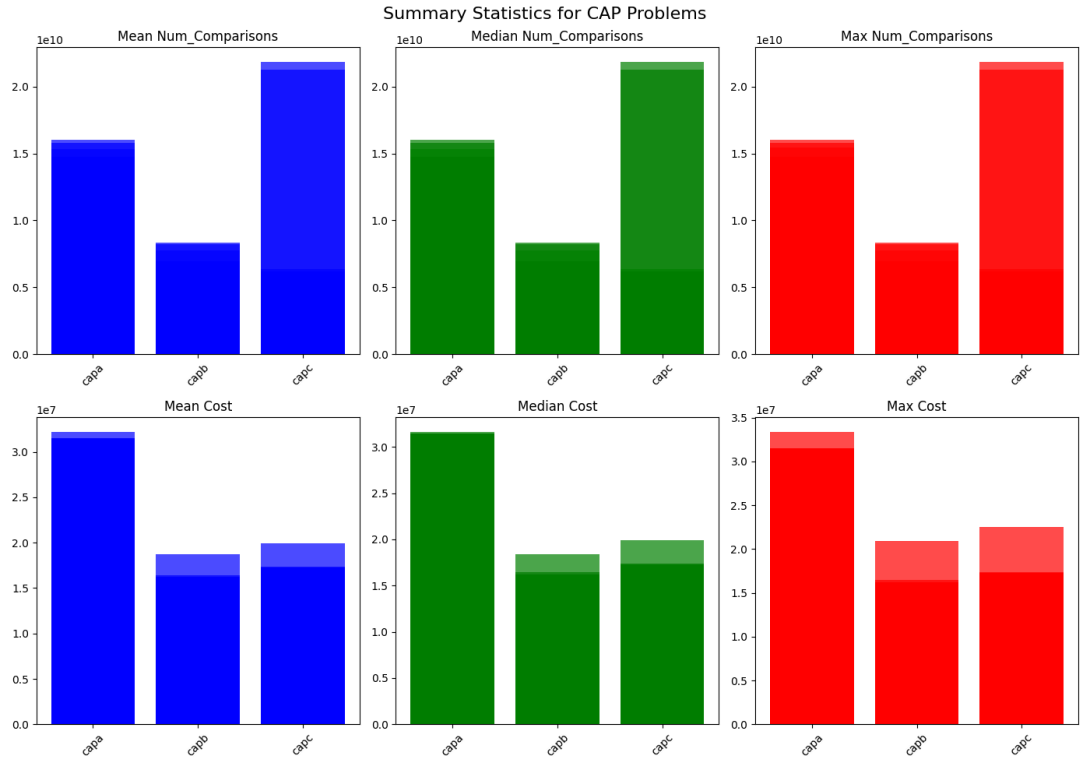


Figura 11: Summary of statistics for Orlibs' problems

Os gráficos fornecidos também apresentam as estatísticas sumárias de um algoritmo Simulated Annealing aplicado a problemas CAP, focando nas métricas de número de comparações e custo. As variantes analisadas são capa, capb e capc.

No que diz respeito ao número de comparações, a variante capc apresenta o maior número médio de comparações, sugerindo uma maior complexidade ou um maior número de iterações necessárias para convergir. A variante capa também tem um número elevado de

comparações, embora inferior a capc, enquanto capb apresenta significativamente menos comparações, indicando uma menor complexidade ou maior eficiência do algoritmo.

A mediana das comparações segue o padrão das médias, com capc no topo, seguida de capa e capb, indicando que os dados não possuem outliers significativos. O máximo de comparações reflete a tendência observada nas médias e medianas, com capc apresentando os valores mais altos e capb os menores, reforçando a hipótese de menor complexidade.

Em termos de custo, o custo médio é maior em capa, indicando que esta variante tende a gerar soluções mais caras. As variantes capb e capc têm custos médios semelhantes, mas inferiores a capa, sugerindo soluções mais eficientes em termos de custo. A mediana do custo para capa é a mais alta, refletindo a tendência observada no custo médio. As variantes capb e capc têm medianas mais baixas, consistentes com os custos médios.

O custo máximo é mais elevado em capa, seguida de capc e capb, confirmando que a variante capa tende a ter instâncias com soluções mais caras, enquanto capb é a mais eficiente em termos de custo.

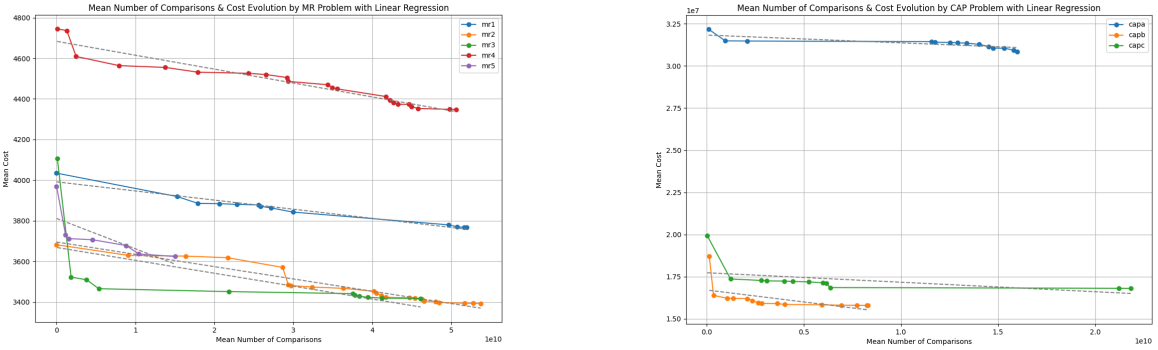


Figura 12 & 13: Mean number of Comparisons & Cost evolution in Cap & MR

Os gráficos apresentados mostram a evolução do número médio de comparações e do custo médio ao longo das iterações de um algoritmo de Simulated Annealing (SA) para dois tipos de problemas, identificados como MR e CAP. Cada gráfico inclui múltiplas execuções do algoritmo (representadas pelas diferentes curvas coloridas), e uma regressão linear pontilhada para cada execução, indicando a tendência geral da redução de custo ao longo das iterações.

No gráfico do Problema MR, observamos que as execuções têm comportamentos distintos. Por exemplo, a curva "mr3" (verde) destaca-se por alcançar um custo médio

significativamente mais baixo mais rapidamente em comparação com as outras execuções. Em contraste, "mr4" (vermelha) mantém um custo médio elevado por mais tempo. Já no gráfico do Problema CAP, todas as execuções mostram uma rápida redução inicial no custo, seguido por uma estabilização. A execução "capb" (laranja) é particularmente eficiente, atingindo custos médios mais baixos de forma consistente. A regressão linear em ambos os gráficos confirma a tendência decrescente geral, evidenciando a eficácia do algoritmo de SA na otimização de custos ao longo do tempo.

3.3 Algoritmo GRASP

3.3.1 História

O GRASP (Greedy Randomized Adaptive Search Procedure) foi desenvolvido no final dos anos 80 e início dos anos 90 por Mauricio G.C. Resende e Celso C. Ribeiro. Criado para resolver problemas de otimização combinatória, o método consiste em duas fases: construção e pesquisa local. Durante a fase de construção, uma solução inicial é criada com decisões parcialmente aleatórias, enquanto a fase de pesquisa local refina essa solução.

No passado, o GRASP rapidamente ganhou popularidade devido à sua simplicidade e eficácia. Foi aplicado a uma vasta gama de problemas de otimização, incluindo roteamento de veículos, alocação de frequências, programação de horários e otimização de redes. As primeiras aplicações mostraram que o GRASP poderia produzir soluções de alta qualidade em tempos de computação razoáveis, mesmo para problemas grandes e complexos. Além disso, a natureza iterativa e paralela do GRASP permitiu que fosse facilmente implementado em sistemas de multiprocessamento, aumentando ainda mais a sua atratividade.

Atualmente, o GRASP continua a ser uma ferramenta popular tanto na investigação académica como em aplicações práticas. As variantes e extensões, como o Path-Relinking, que procura melhorar ainda mais a solução ao explorar caminhos entre soluções elite, e versões híbridas que combinam GRASP com outras meta-heurísticas, ampliaram o seu espectro de aplicação. Os investigadores continuam a desenvolver novas variantes do GRASP e a explorar as suas aplicações em novos domínios, como bioinformática, logística e design de circuitos integrados. O GRASP é frequentemente utilizado em conjunto com outras técnicas de otimização, como algoritmos genéticos e Simulated Annealing, para criar métodos híbridos que combinam os pontos fortes de diferentes abordagens. Esses métodos híbridos têm demonstrado ser particularmente eficazes em resolver problemas altamente complexos e dinâmicos.

O futuro do GRASP parece promissor, com várias direções de investigação emergentes. Uma área de interesse crescente é a integração do GRASP com técnicas de aprendizagem automática. Por exemplo, algoritmos de aprendizagem podem ser usados para ajustar os parâmetros do GRASP em tempo real ou para prever as melhores estratégias de construção e *Local Search* através das características do problema. Outra tendência é a aplicação do GRASP em problemas de otimização multiobjetivo, onde várias funções objetivo devem ser otimizadas simultaneamente. A capacidade do GRASP de explorar eficientemente o espaço de soluções faz dele uma ferramenta ideal para este tipo de problema.

Finalmente, o avanço da computação quântica pode abrir novas possibilidades para o GRASP. Algoritmos quânticos têm o potencial de acelerar significativamente a fase de pesquisa local do GRASP, permitindo que ele resolva problemas ainda maiores e mais complexos.

Resumindo, o GRASP evoluiu significativamente desde a sua criação e continua a ser uma ferramenta valiosa na caixa de ferramentas de otimização combinatória. O seu futuro parece brilhante, com muitas oportunidades para inovar e ser aplicado em novos problemas.

3.3.2 Introdução do método relativamente ao UFLP

A introdução do GRASP no Uncapacitated Facility Location Problem (UFLP) trouxe melhorias significativas. O UFLP, que visa minimizar os custos de abertura de instalações e de transporte, beneficiou-se enormemente da combinação de heurísticas gulosas e elementos aleatórios do GRASP.

Na fase de construção do GRASP, são considerados tanto os custos fixos como os custos variáveis para criar soluções iniciais diversificadas. Esta fase é iterativa e utiliza um critério guloso para seleccionar componentes da solução, introduzindo simultaneamente uma componente aleatória para garantir diversidade e evitar soluções locais ótimas prematuras. A fase de *Local Search* refina então essas soluções iniciais, explorando a vizinhança das soluções para identificar possíveis melhorias. Estudos empíricos têm mostrado que o GRASP supera muitos métodos tradicionais quando aplicado ao UFLP, especialmente quando são introduzidas inovações como o Path-Relinking, que melhora ainda mais a qualidade das soluções ao explorar caminhos entre soluções elite.

3.3.3 Descrição do Algoritmo

O GRASP é uma meta-heurística que combina técnicas gulosas e aleatórias para resolver problemas de otimização combinatória em duas fases: construção e pesquisa local. Na fase de construção, uma solução inicial é criada iterativamente. Em cada iteração, componentes da solução são seleccionados com base num critério guloso que avalia a melhor escolha local, mas é adicionada uma componente aleatória que permite a diversificação das soluções. Este equilíbrio entre ganância e aleatoriedade ajuda a explorar o espaço de soluções de maneira eficaz e a evitar ficar preso em ótimos locais.

Uma vez que a solução inicial é construída, ela é submetida à fase de pesquisa local. Nesta fase, a solução é iterativamente melhorada explorando os seus vizinhos imediatos, que consiste em todas as soluções que podem ser alcançadas com pequenas modificações na solução atual. Se uma solução melhor é encontrada na vizinhança, ela substitui a solução atual. Este processo continua até que nenhuma melhoria adicional possa ser encontrada, indicando que um ótimo local foi alcançado.

O GRASP é particularmente eficaz devido à sua natureza iterativa e adaptativa. Cada execução do algoritmo pode produzir uma solução diferente, permitindo a exploração de uma grande parte do espaço de soluções possíveis. Além disso, o GRASP pode ser facilmente paralelo, com múltiplas soluções sendo construídas e refinadas em paralelo, aumentando a eficiência computacional. A combinação destas características permite ao GRASP encontrar soluções de alta qualidade de forma eficiente e robusta, tornando-o uma ferramenta valiosa para resolver problemas complexos de otimização combinatória como o UFLP.

3.3.4 Modelo Matemático

O algoritmo GRASP consiste em duas fases principais: a fase de construção e a fase de *Local Search*. Vamos definir matematicamente cada uma dessas fases no contexto do UFLP.

Parâmetros:

- N **:** Número de clientes.
- M **:** Número de instalações.
- f_i **:** Custo fixo para abrir a instalação i.
- c_{ij} **:** Custo para alocar o cliente j à instalação i.

Variáveis de Decisão:

- $y_i \in \{0, 1\}$: Indica se a instalação i está aberta $(y_i = 1)$ ou não $(y_i = 0)$.
- $x_{ij} \in \{0, 1\}$: Indica se o cliente j está alocado à instalação i $(x_{ij} = 1)$ ou não $(x_{ij} = 0)$.

Função Objetivo:

$$Minimizar\ Z = \sum_{i=1}^{M} f_i y_i + \sum_{i=1}^{M} \sum_{j=1}^{N} c_{ij} x_{ij}$$

Restrições:

$$\sum_{i=1}^{M} x_{ij} = 1\ ,\ \forall_j \in \{1, ..., N\}$$

$$x_{ij} \leq y_i\ ,\ \forall\ i \in \{1, ..., M\}\ ,\ \forall\ j \in \{1, ..., N\}$$

$$x_{ij}\ , y_i \in \{0, 1\}, \forall i, j$$

3.3.4.1. Fase de Construção do GRASP

Na fase de construção, uma solução inicial é criada iterativamente com decisões parcialmente aleatórias, garantindo que a solução não seja meramente uma sequência de decisões gulosas.

1. **Inicialização:**
   - Inicializa-se todas as instalações fechadas $(y_i = 0)$.
   - Inicializa-se todas as variáveis de alocação $(x_{ij} = 0)$.
2. **Lista Restrita de Candidatos (RCL):**
   - Para cada instalação i, calcula-se o custo de abri-la e de alocar os clientes a mesma.
   - Define-se a lista restrita de candidatos (RCL) com as melhores instalações potenciais para abrir, com base no critério modificado pela aleatoriedade.

$$RCL = \{\, i \mid custo(i) \leq limite\}$$

Onde:

$$limite = custo\_min + \alpha\,(custo\_max - custo\_min)$$

3. **Seleção Aleatória da RCL:**
   - Seleciona-se aleatoriamente uma instalação i da RCL para abrir $(y_i = 1)$.
   - Distribuem-se os clientes à nova instalação aberta minimizando os custos de alocação.
4. **Repetição:**
   - Repetem-se os passos acima até que uma solução viável completa seja construída.

3.3.4.2. Fase de Local Search

Na fase de *Local Search*, a solução inicial é refinada explorando a vizinhança da solução atual para encontrar melhorias.

1. **Reassign Customers:**

- Distribuem-se os clientes para as instalações abertas minimizando o custo total.

$$custo\_alocação\ (j) = i \in min\ c_{ij}$$

$$custo\_total = \sum_{i \in intalações\ abertas} f_i + \sum_{j=1}^{N} custo\_alocação(j)$$

2. **CloseWarehouse:**
   - Para cada instalação aberta i, fecha-se temporariamente $(y_i = 0)$ e distribuem-se os clientes, se o custo total diminuir, é mantida a instalação fechada.
3. **OpenWarehouse:**
   - Para cada instalação fechada i, abre-se temporariamente $(y_i = 1)$ e distribuem-se os clientes, se o custo total diminuir, é mantida a instalação aberta.
4. **OpenCloseWarehouse:**
   - Abre-se uma instalação fechada i e fecha-se uma instalação aberta j, se o custo total diminui, são mantidas as mudanças.
5. **Iteração até Convergência:**
   - O processo apenas é interrompido se não forem encontradas melhorias (solução ótima local)

3.3.5 Desempenho e Complexidade

O GRASP é uma abordagem eficaz para o UFLP, equilibrando a qualidade das soluções e o tempo de execução, especialmente útil em problemas de grande escala onde métodos exatos são impraticáveis.

3.3.5.1 Desempenho do GRASP no UFLP

Qualidade das Soluções:

As soluções produzidas pelo GRASP são frequentemente de alta qualidade e comparáveis às obtidas por outras técnicas heurísticas. A abordagem combinada de construção e *Local Search* permite encontrar soluções eficientes para o problema de localização de instalações.

Velocidade de Convergência:

O GRASP tende a encontrar boas soluções rapidamente, especialmente nos estágios iniciais das iterações. A fase construtiva gera soluções iniciais que são aprimoradas rapidamente na fase de *Local Search,* resultando em uma convergência acelerada para soluções de alta qualidade.

3.3.5.2 Complexidade do GRASP no UFLP

Fase Construtiva:

A construção de uma solução inicial no GRASP tem uma complexidade de $O(mn)$, onde m é o número de instalações e n é o número de clientes. Este processo envolve a seleção aleatória e a avaliação dos custos de abrir instalações e atribuir clientes, sendo eficiente para grandes instâncias do problema.

Iterações do GRASP:

A complexidade total do GRASP é proporcional ao número de iterações definidas pelo utilizador multiplicado pela complexidade combinada das fases construtiva e de *Local Search*. Se k for o número de iterações, a complexidade total será $O(k \cdot mn)$. A escolha adequada do número de iterações é crucial para equilibrar a qualidade das soluções e o tempo de execução.

3.3.6 Vantagens & Desvantagens

3.3.6.1 Vantagens

1. **Qualidade das Soluções**: Produz soluções frequentemente de alta qualidade, comparáveis ou superiores às obtidas por outras técnicas heurísticas.
2. **Flexibilidade**: Pode ser adaptado para diferentes problemas de otimização, com parâmetros ajustáveis como o tamanho da Lista Restrita de Candidatos (RCL) e a intensidade da *Local Search*.
3. **Facilidade de Implementação**: Relativamente fácil de implementar, com uma estrutura simples composta por uma fase construtiva seguida de uma *Local Search*.

4. **Robustez**: Capaz de escapar de mínimos locais devido à sua abordagem iterativa e adaptativa, explorando eficientemente o espaço de soluções.
5. **Integração com Outras Técnicas**: Pode ser combinado com outras meta-heurísticas, como algoritmos genéticos ou tabu search, para melhorar a qualidade das soluções e a eficiência do algoritmo.

3.3.6.2 Desvantagens

1. **Complexidade Computacional**: A complexidade $O(k \cdot mn)$ pode ser elevada para grandes instâncias do problema, resultando em tempos de execução longos.
2. **Dependência de Parâmetros**: O desempenho é sensível à escolha dos parâmetros, como o tamanho da RCL e o número de iterações, exigindo experimentação e ajuste fino.
3. **Qualidade das Soluções Iniciais**: Soluções iniciais de baixa qualidade na fase construtiva podem influenciar negativamente o desempenho, dificultando a obtenção de soluções ótimas.
4. **Local Search Dependente de Implementação**: A eficiência da *Local Search* depende fortemente da implementação específica e das estratégias utilizadas, podendo levar a um desempenho subótimo se mal projetadas.
5. **Possibilidade de Convergência Lenta**: Em problemas muito complexos, o GRASP pode convergir lentamente, necessitando de muitas iterações para encontrar soluções de alta qualidade, o que pode ser impraticável em contextos com restrições de tempo.

3.3.7 Aplicações no mundo Real

O GRASP é uma técnica versátil e eficiente, aplicável a uma ampla gama de problemas no mundo real. Estes são apenas alguns exemplos:

- **Problema de Localização de Instalações:** Determinação da localização ideal de depósitos, centros de distribuição ou fábricas para minimizar os custos de transporte e operação.
- **Problema do Caixeiro Viajante (TSP):** Planeamento de rotas ótimas para veículos de entrega para minimizar a distância ou o tempo de viagem.
- **Desenho de Redes de Telecomunicações:** Planeamento da estrutura de redes de comunicação para otimizar a cobertura e minimizar os custos de instalação.
- **Planeamento da Produção:** Otimização do cronograma de produção em fábricas para maximizar a eficiência e minimizar os tempos de inatividade das máquinas.

- **Problema de Corte de Stock:** Otimização do corte de materiais em peças menores para minimizar o desperdício.
- **Programação de Tarefas:** Planeamento e alocação de recursos em projetos complexos para garantir a conclusão no menor tempo possível ou dentro do orçamento.
- **Problema do Caminho Crítico:** Identificação das atividades críticas em um projeto para evitar atrasos e melhorar a gestão do cronograma.
- **Desenho de Redes de Distribuição de Energia:** Planeamento da rede de distribuição de energia elétrica para minimizar perdas e garantir a confiabilidade.
- **Otimização de Centrais Elétricas:** Planeamento da operação de centrais elétricas para maximizar a produção e minimizar os custos operacionais.
- **Planeamento de Turnos de Enfermagem:** Otimização dos horários de trabalho dos enfermeiros para garantir a cobertura adequada e minimizar a sobrecarga de trabalho.
- **Distribuição de Medicamentos:** Planeamento da logística de distribuição de medicamentos para garantir a disponibilidade e minimizar os custos de transporte.
- **Problemas de Alocação de Capital:** Distribuição eficiente de recursos financeiros entre diferentes projetos ou unidades de negócio.

3.3.8 Análise de Resultados

Nessa seção, analisamos os resultados obtidos com a implementação do algoritmo GRASP para a resolução do problema de alocação de clientes a armazéns. O algoritmo foi configurado para utilizar uma lista restrita dinâmica durante a fase construtiva, permitindo uma seleção adaptativa dos candidatos para abertura de armazéns com base em um critério aleatório. Além disso, o critério de paragem da *Local Search* foi definido de maneira que a pesquisa só cessasse quando não fosse possível melhorar mais o custo total da solução.

Utilizamos um valor de $\alpha = 0$ para manter um limiar explorativo, permitindo uma análise do comportamento do algoritmo na *Local Search*. Este valor garante que a lista de candidatos restritos inclua todos os possíveis armazéns, promovendo uma maior diversidade na solução inicial.

Agora, procederemos com uma análise detalhada das tabelas e figuras a seguir, a fim de compreender melhor o comportamento do algoritmo GRASP nas diferentes instâncias

testadas. Esta análise visual e tabular permitirá identificar padrões, avaliar a eficiência das soluções encontradas e comparar o desempenho do algoritmo em diversos cenários.

Instance	Mean	median	Min (best)	Max (worst)	STD Deviation	Number of Comparisons	Optimal Solution
capa	*9.16197e+07*	*8.85784e+07*	*1.171565e+07*	*1.82644e+08*	*4.9645e+07*	*1475674100*	*17156454.478*
capb	*4.09764e+07*	*3.91595e+07*	*1.29791e+07*	*7.66358e+07*	*1.92825e+06*	*1794717093*	*12979071.582*
capc	*3.11902e+07*	*2.9709e+07*	*1.15353e+07*	*5.59415e+07*	*1.31874e+06*	*1735333514*	*11505594.329*
mr1	*83025.2*	*80045*	*2349.86*	*177707*	*51347.6*	*66576773903*	*2349.856*
mr2	*77140.2*	*75251.3*	*2344.76*	*162773*	*46961.5*	*66670416168*	*2344.757*
mr3	*73410.1*	*70243.1*	*2183.23*	*159082*	*46199.7*	*66908044753*	*2183.235*
mr4	*88252.8*	*85901.6*	*2433.11*	*186877*	*53806.4*	*66648640380*	*2433.110*
mr5	*83439.4*	*80688.2*	*2344.35*	*177636*	*51383.7*	*66420664909*	*2344.353*

Tabela 3: Resultados do Algoritmo GRASP em diferentes Instâncias do UFLP

A tabela 3 apresenta os resultados do algoritmo GRASP aplicados a diversas instâncias do UFLP. As métricas avaliadas incluem a média, mediana, valor mínimo (melhor), valor máximo (pior) e o desvio padrão das soluções obtidas para cada instância. As instâncias são rotuladas como *capa, capb, capc, mr1, mr2, mr3, mr4 e mr5*.

Análise das Instâncias "capa", "capb" e "capc":

capa

A instância “capa” apresenta a maior média (9.16197e+07) entre todas as instâncias, indicando que, em geral, as soluções encontradas são mais caras. O desvio padrão relativamente alto (4.9645e+07) sugere uma grande variabilidade nas soluções, o que pode indicar que o algoritmo explorou uma ampla gama de possíveis soluções.

capb

A instância “capb” tem uma média menor que capa (4.09764e+07), indicando soluções geralmente mais económicas. O desvio padrão (1.92825e+07) é também menor que “capa”, mas ainda significativo, apontando para uma variação notável nas soluções.

capc

A instância “capc” apresenta a menor média (3.11902e+07) e desvio padrão (1.31874e+06) entre as instâncias de “capa”, sugerindo soluções mais estáveis e geralmente menos dispendiosas. De todas as instâncias (incluindo as instâncias “mr”) é a única em que o algoritmo foi incapaz de encontrar a solução ótima.

Instâncias mr1, mr2, mr3, mr4 e mr5

mr1

A instância “mr1” mostra uma alta variabilidade com um desvio padrão significativo (51347.6). A diferença entre o valor mínimo (2349.86) e máximo (177707) é grande, indicando que o algoritmo encontrou tanto soluções muito boas quanto soluções relativamente fracas.

mr2

A instância “mr2” tem resultados semelhantes ao “mr1”, mas com uma menor variabilidade (46961.5), sugerindo soluções mais consistentes. A média é ligeiramente menor que a de “mr1” (77140.2 vs 83025.2).

mr3

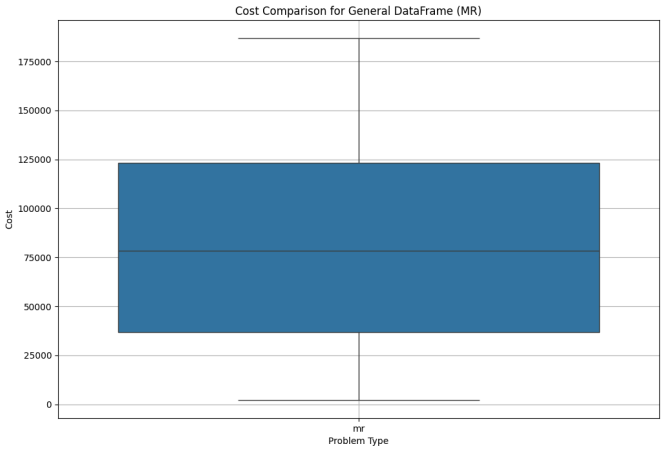
A instância “mr3” apresenta a menor média (73410.1) entre as instâncias de “mr”, indicando soluções geralmente mais económicas. O desvio padrão (46199.7) também é relativamente alto, refletindo uma alta variabilidade nas soluções.

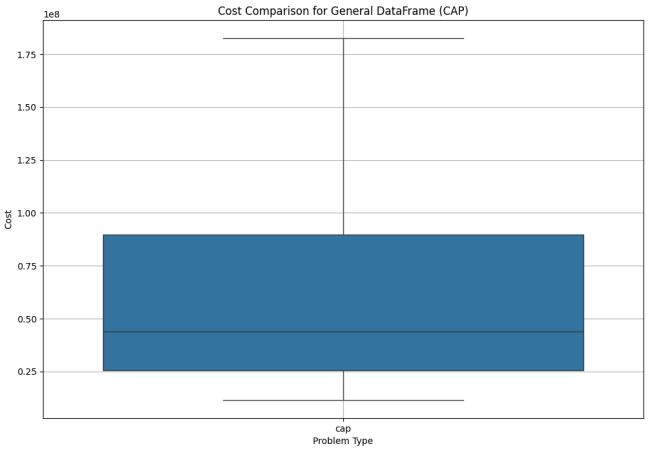
mr4

A instância mr4 tem a maior média (88252.8) e a maior variabilidade, com o maior desvio padrão (53806.4), indicando uma ampla gama de soluções encontradas pelo algoritmo.

mr5

A instância mr5 possui uma média elevada (83439.4), sugerindo soluções geralmente dispendiosas. O desvio padrão também é alto (51383.7), indicando variabilidade significativa nas soluções.





Figuras 14 & 15: Cost Comparison for MR & CAP

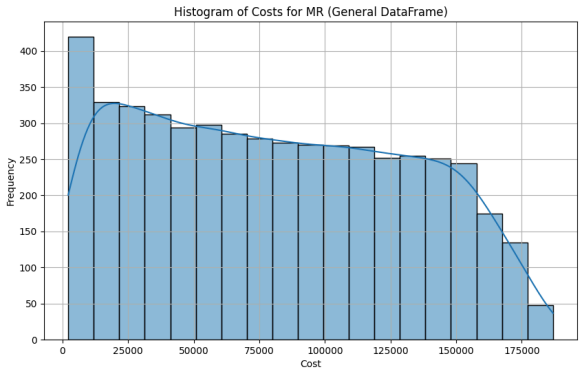
Através da análise das figuras fornecidos, é possível tirar algumas conclusões sobre o desempenho do algoritmo GRASP nas diferentes instâncias CAP e MR.

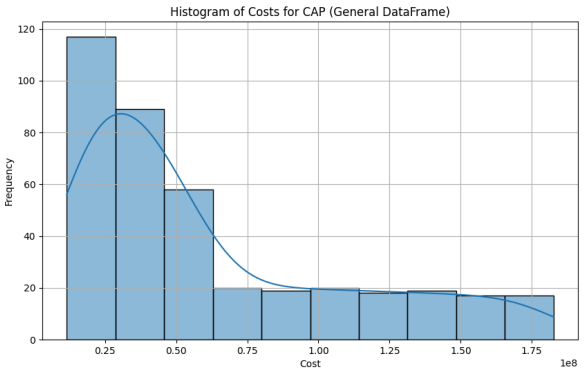
Variação dos Custos

Verificamos que a variação de custos é significativamente superior nas instâncias CAP em comparação com as instâncias MR. Isso é evidenciado pelos gráficos de caixa que mostram uma dispersão muito maior nos custos das instâncias CAP, com valores que se estendem até 1.8e+08, enquanto as instâncias MR apresentam uma variação bem mais contida, com valores máximos na faixa de 1.77e+05.

Comparação das Medianas

Para os ”MR”, a mediana está em torno de 100,000 com uma ampla variação, indicando alta dispersão e presença de outliers. Para os “CAP”, a mediana é de 75 milhões, também com grande variação e custos médios muito mais altos, refletindo maior complexidade. Ambos os problemas mostram alta variabilidade nos custos, sugerindo influência de diversos fatores como tamanho dos dados e eficiência do algoritmo.





Figuras 16 & 17: Histogram of Costs for MR & CAP

Os histogramas apresentados mostram a distribuição dos custos para os problemas “MR” e “CAP” I.

No histograma dos “MR” (à esquerda), observamos que a maioria dos custos está concentrada na faixa de 0 a 50,000, com uma frequência decrescente à medida que o custo aumenta. Isso sugere que, para a maioria dos casos, os custos são relativamente baixos e mais próximos entre si. A distribuição dos custos é mais uniforme, indicando menor variabilidade e permitindo que o algoritmo atinja valores de baixo custo mais rapidamente.

Por outro lado, no histograma dos “CAP” (à direita), vemos uma alta concentração de custos na faixa de 0 a 0.5x10^8, mas com uma cauda longa que se estende até 1.75x10^8. Isso indica uma maior discrepância nos resultados de custos para o problema “CAP”, com alguns casos sendo significativamente mais caros. A variabilidade maior sugere que o algoritmo tem mais dificuldade em alcançar consistentemente valores de baixo custo, resultando em uma distribuição mais dispersa e menos previsível.

Essas observações indicam que, enquanto os problemas “MR” apresentam custos mais próximos e consistentes, permitindo uma otimização mais eficaz, os problemas “CAP” sofrem com uma maior dispersão de custos, dificultando a obtenção de soluções de baixo custo de forma rápida e consistente.

3.4 Algoritmo Hill Climbing

3.4.1 História

O algoritmo Hill Climbing emergiu como uma técnica fundamental na área de otimização heurística durante as décadas de 1950 e 1960, uma época em que a computação começou a ganhar um papel significativo na resolução de problemas complexos. A conceção inicial do HC foi motivada pela necessidade de encontrar soluções localmente ótimas através de mudanças incrementais, representando um avanço importante na procura de soluções para problemas onde determinar uma solução ideal era particularmente desafiante. Durante esse período, a simplicidade e a eficiência do algoritmo destacaram-se, especialmente em cenários onde a pesquisa exaustiva era impraticável devido ao alto custo computacional.

No passado, o HC tornou-se uma ferramenta popular por adotar uma abordagem iterativa. A solução atual é continuamente melhorada pela avaliação de vizinhos próximos e a escolha do melhor entre eles. Essa simplicidade e eficácia fizeram do algoritmo uma escolha popular para diversos problemas de otimização, como a alocação de recursos e o planeamento de rotas. No entanto, as limitações tornaram-se rapidamente aparentes, especialmente em relação ao problema dos mínimos locais, onde o algoritmo podia ficar preso a uma solução subótima.

Para superar essas limitações, o HC evoluiu ao longo das décadas subsequentes. Variantes como o HC Estocástico introduziram elementos de aleatoriedade no processo de pesquisa, permitindo ao algoritmo escapar de mínimos locais e explorar mais eficazmente o espaço de soluções. Outras adaptações significativas incluem o Simulated Annealing, que combina os princípios do HC com estratégias de arrefecimento para proporcionar uma exploração mais robusta do espaço de solução.

No presente, o HC e as suas variantes continuam a ser estudadas e aplicadas em diversas áreas da computação e otimização. A pesquisa atual foca em melhorar a eficiência e a eficácia do algoritmo, adaptando-o a problemas cada vez mais complexos e de maior dimensão. A implementação de técnicas híbridas, que combinam o HC com outros métodos de otimização, tem mostrado progressos significativos em melhorar a capacidade de encontrar soluções globais ótimas.

Quanto ao futuro, espera-se que o Hill Climbing continue a evoluir, incorporando avanços em inteligência artificial e *Machine Learning*. O desenvolvimento de algoritmos mais adaptativos e inteligentes, capazes de ajustar dinamicamente as estratégias de pesquisa com

base no problema específico, é uma área com futuro. Além disso, a integração do HC com tecnologias emergentes, como computação quântica, pode abrir novas possibilidades para resolver problemas de otimização que atualmente são intratáveis.

A evolução contínua e a adaptação do Hill Climbing às necessidades modernas de computação destacam a importância duradoura como uma técnica essencial de otimização heurística. Essa capacidade de transformação e melhoria contínua assegura que o Hill Climbing continuará a ser relevante e eficaz para as gerações futuras de investigadores e profissionais da computação.

3.4.2 Introdução do método relativamente ao UFLP

O algoritmo Hill Climbing oferece uma abordagem heurística eficaz para resolver o UFLP. Este problema envolve a determinação de um conjunto de locais para instalar unidades de serviço, como armazéns ou centros de distribuição, de forma a minimizar o custo total de operação, que inclui tanto os custos fixos de instalação como os custos variáveis de atendimento aos clientes.

No contexto do UFLP, o HC inicia com uma solução inicial, onde todas as instalações possíveis podem estar abertas ou fechadas. A partir desta solução inicial, o algoritmo procede através de ajustes iterativos, onde se examinam as soluções vizinhas, alterando o estado de uma ou mais instalações (aberta para fechada ou vice-versa) para encontrar uma solução que melhore o critério de otimização. Este processo iterativo continua até que nenhuma melhoria adicional seja possível, indicando que um ótimo local foi alcançado.

3.4.3 Descrição do Algoritmo

O algoritmo Hill Climbing é uma técnica heurística de otimização que visa encontrar uma solução localmente ótima para um problema, começando com uma solução inicial e realizando pequenas alterações incrementais. Este método baseia-se numa abordagem de *Local Search*, onde o foco está em explorar as soluções vizinhas à solução atual, movendo-se sempre na direção que melhora o valor da função objetivo.

O processo começa com a escolha de uma solução inicial, que pode ser gerada aleatoriamente ou baseada em algum critério heurístico. A partir desta solução, o algoritmo

avalia as soluções vizinhas, que são obtidas através de pequenas modificações na solução atual. A natureza destas modificações depende do problema específico e da definição de vizinhança escolhida.

A cada iteração, o algoritmo compara o valor da função objetivo da solução atual com os valores das soluções vizinhas. Se uma solução vizinha apresenta um valor melhor (ou seja, menor custo, maior lucro, etc.), esta solução vizinha torna-se a nova solução atual. Este processo iterativo continua até que não se encontre nenhuma solução vizinha que melhore a função objetivo, indicando que um ótimo local foi alcançado.

No entanto, devido à sua natureza de busca local, o HC pode ficar preso em mínimos locais, onde nenhuma das soluções vizinhas melhora a solução atual, mesmo que existam soluções melhores em outras áreas do espaço de solução. Para mitigar este problema, várias variantes e técnicas adicionais podem ser integradas ao algoritmo HC, tais como:

- **Hill Climbing Estocástico**: Introduz elementos de aleatoriedade no processo de seleção de vizinhos. Em vez de escolher sempre a melhor solução vizinha, o algoritmo pode ocasionalmente selecionar uma solução vizinha aleatória, permitindo a exploração de novas regiões do espaço de solução e evitando ficar preso em mínimos locais.
- **Perturbações Controladas**: Introdução de perturbações conscientes na solução atual para "sacudir" o algoritmo de um mínimo local. Estas perturbações podem envolver mudanças significativas na solução atual para explorar novas áreas do espaço de solução.
- **Reinicializações Aleatórias**: Periodicamente, a solução atual é substituída por uma nova solução inicial gerada aleatoriamente. Esta técnica permite ao algoritmo escapar de mínimos locais e explorar diferentes regiões do espaço de solução.
- **Intensificação e Diversificação**: Estratégias de intensificação focam em explorar profundamente os vizinhos de soluções promissoras, enquanto estratégias de diversificação incentivam a exploração de regiões menos investigadas do espaço de solução. Esta abordagem ajuda a equilibrar a exploração e a intensificação, aumentando as chances de encontrar a solução globalmente ótima.

A eficácia do algoritmo HC depende significativamente da escolha de uma estratégia adequada e de critérios de paragem bem definidos. Os critérios de paragem podem incluir um número máximo de iterações, um tempo limite de execução, ou a ausência de melhorias na função objetivo após um certo número de iterações.

Em suma, o algoritmo HC é uma técnica poderosa e simples para resolver problemas de otimização, com a vantagem de ser facilmente implementado e adaptável a diversos tipos

de problemas. No entanto, a sua eficácia pode ser limitada pela tendência de ficar preso em mínimos locais, o que pode ser mitigado através da incorporação de variantes e técnicas avançadas.

3.4.4 Modelo Matemático

O algoritmo Hill Climbing é um método de otimização iterativo que procura melhorar uma solução inicial movendo-se para uma solução vizinha que tenha um custo menor. No contexto do UFLP, o modelo matemático e o algoritmo podem ser definidos da seguinte maneira:

Parâmetros:

- N é o número de clientes.
- M é o número de instalações.
- f_i é o custo fixo para abrir a instalação i.
- c_{ij} é o custo para alocar o cliente j à instalação i.

Variáveis de Decisão:

- $y_i \in \{0, 1\}$: Indica se a instalação i está aberta $(y_i = 1)$ ou não $(y_i = 0)$.
- $x_{ij} \in \{0, 1\}$: Indica se o cliente j está alocado à instalação i $(x_{ij} = 1)$ ou não $(x_{ij} = 0)$.

Função Objetivo:

$$Minimizar\ Z = \sum_{i=1}^{M} f_i y_i + \sum_{i=1}^{M} \sum_{j=1}^{N} c_{ij} x_{ij}$$

Restrições:

1. Cada cliente deve ser alocado a exatamente uma instalação:

$$\sum_{i=1}^{M} x_{ij} = 1\ ,\ \forall_j \in \{1, ..., N\}$$

2. Um cliente só pode ser alocado a uma instalação se esta estiver aberta:

$$x_{ij} \leq y_i \,,\ \forall\ i \in \{1, ..., M\} \,, \forall\ j \in \{1, ..., N\}$$

3. As variáveis de decisão são binárias:

$$x_{ij} \,, y_i \in \{0, 1\}, \forall i, j$$

3.4.4.1. Fase de Construção do Hill Climbing

Na fase de construção do Hill Climbing, uma solução inicial é criada, que servirá como ponto de partida para o processo de melhoria iterativa. O objetivo é obter uma solução viável rapidamente, ainda que não seja ótima.

1. **Inicialização:**
    1. Inicializa-se todas as instalações fechadas $(y_i = 0)$.
    2. Inicializa-se todas as variáveis de alocação $(x_{ij} = 0)$.
2. **Solução Inicial:**
    1. Escolhe-se uma solução inicial, onde algumas instalações são abertas aleatoriamente
    2. Os clientes são alocados a essas instalações de forma a minimizar os custos iniciais.

3.4.4.2. Fase de Local Search do Hill Climbing

Na Fase de *Local Search*, o algoritmo Hill Climbing pesquisa refinar a solução inicial explorando a vizinhança da solução atual para encontrar melhorias no custo total. A estratégia visa mover-se para soluções vizinhas que ofereçam um custo menor.

1. **Reatribuição de Clientes**:
    - Para cada cliente j:
        - Verifica-se se existe uma instalação aberta i diferente da atualmente alocada que ofereça um custo de alocação c_{ij} menor.
        - Se encontrado, o cliente i é realocado para essa nova instalação i.
        - Após cada redistribuição, o custo total da solução é recalculado.

2. **Movimentos Locais**:
    - **Fechar Instalação**:
        - Para cada instalação aberta i:
            - Fecha-se temporariamente a instalação (y_{ij} = 0) e redistribuem-se os clientes para as demais instalações abertas.
            - Se o custo total diminuir com a instalação fechada, mantém-se a instalação fechada na solução atual.
    - **Abrir Instalação**:
        - Para cada instalação fechada i:
            - Abre-se temporariamente a instalação (y_{ij} = 1) e redistribuem-se os clientes considerando esta nova instalação aberta.
            - Se o custo total diminuir com a instalação aberta, mantém-se a instalação aberta na solução atual.
    - **Troca de Instalação**:
        - Abre-se uma instalação fechada i e fecha-se uma instalação aberta j.
        - Redistribuem-se os clientes de acordo com as novas condições de abertura e fechamento das instalações.
        - Se o custo total diminuir com esta troca de instalação, mantêm-se as mudanças na solução atual.
3. **Iteração até Convergência**:
    - O processo de *Local Search* é repetido iterativamente até que nenhuma melhoria adicional seja encontrada na vizinhança da solução atual.
    - Isso indica que um máximo local foi alcançado, e o algoritmo para nesta etapa.
4. **Critério de Paragem**:
    - Não existem mais melhorias possíveis na solução atual.
    - Alternativamente, pode ser definido um número máximo de iterações sem melhoria como critério de paragem.

3.4.5 Desempenho e Complexidade

O Hill Climbing é uma abordagem eficaz para o UFLP, equilibrando a qualidade das soluções e o tempo de execução, especialmente útil em problemas de grande escala onde métodos exatos são impraticáveis.

3.4.5.1 Desempenho do Hill Climbing no UFLP

Qualidade das Soluções:

As soluções produzidas pelo Hill Climbing variam em qualidade, dependendo do ponto de partida e da paisagem do espaço de soluções. Embora possa não alcançar a solução máxima global, o Hill Climbing frequentemente encontra soluções satisfatórias que são competitivas com outras técnicas heurísticas. Esta abordagem iterativa é eficaz para ajustar e melhorar soluções para o problema de localização de instalações.

Velocidade de Convergência:

O Hill Climbing tende a convergir rapidamente para um máximo local. Nos estágios iniciais das iterações, a melhoria das soluções é rápida, pois cada passo tende a levar a uma solução melhor. Contudo, a convergência para um máximo local pode ocorrer antes de encontrar a melhor solução possível, especialmente se a função de custo tiver muitos picos e vales.

3.4.5.2 Complexidade do Hill Climbing no UFLP

Fase de Melhoria Iterativa:

A melhoria iterativa de uma solução no Hill Climbing tem uma complexidade de $O(mn)$, onde m é o número de instalações e n é o número de clientes. Este processo envolve a seleção e avaliação de movimentos locais, ajustando a solução atual para melhorar o custo total de abrir instalações e atribuir clientes.

Iterações do Hill Climbing:

A complexidade total do Hill Climbing é proporcional ao número de iterações definidas pelo utilizador multiplicado pela complexidade da fase de melhoria iterativa. Se k for o número de iterações, a complexidade total será $O(k \cdot mn)$. A escolha adequada do número de iterações é crucial para equilibrar a qualidade das soluções e o tempo de execução. A abordagem pode ser complementada com técnicas como reinicialização aleatória ou diferentes estratégias de vizinhança para escapar de máximos locais e explorar melhor o espaço de soluções.

3.4.6 Vantagens & Desvantagens

3.4.6.1 Vantagens

1. **Qualidade das Soluções:** Hill Climbing pode produzir soluções de boa qualidade que são competitivas com outras heurísticas, especialmente para problemas de otimização com funções de custo bem comportadas.
2. **Simples de Implementar:** A implementação do Hill Climbing é direta, envolvendo apenas a melhoria iterativa de uma solução inicial até que não haja mais melhorias globais possíveis.
3. **Rapidez:** Geralmente, Hill Climbing encontra soluções rapidamente, sendo eficiente em termos de tempo, principalmente em problemas de pequena a média escala.
4. **Baixo Requisito de Memória:** A abordagem requer pouca memória, pois só mantém a solução atual e a vizinhança imediata.
5. **Adaptabilidade:** Pode ser facilmente adaptado e combinado com outras heurísticas e técnicas de otimização, como reinicialização aleatória para escapar de máximos locais.

3.4.6.2 Desvantagens

1. **Convergência em Máximos Locais:** Hill Climbing frequentemente converge para máximos locais, especialmente em paisagens de solução com muitos picos e vales, o que pode impedir a descoberta da solução máxima global.
2. **Dependência da Solução Inicial:** A qualidade da solução final é altamente dependente da solução inicial, o que pode levar a resultados inconsistentes.
3. **Falta de Diversidade:** O método não explora adequadamente o espaço de soluções, podendo ficar preso em áreas limitadas do espaço de pesquisa.
4. **Necessidade de Variação:** Para resolver problemas complexos, muitas vezes é necessário introduzir variações, como a reinicialização aleatória ou a implementação de técnicas adicionais para evitar mínimos locais.
5. **Escalabilidade:** Embora rápido para problemas menores, o Hill Climbing pode não ser eficiente para problemas de grande escala, pois pode necessitar de muitas reinicializações para encontrar boas soluções.

3.4.7 Aplicação no mundo Real

O Hill Climbing é uma técnica prática e eficiente para vários problemas de otimização do mundo real. Alguns exemplos incluem:

- **Problema de Localização de Instalações:** Determinação de locais para depósitos, centros de distribuição ou fábricas para reduzir custos operacionais.
- **Problema do Caixeiro Viajante (TSP):** Planeamento de rotas eficientes para veículos de entrega, minimizando distâncias ou tempos de viagem.

- **Desenho de Redes de Telecomunicações:** Planeamento da infraestrutura de redes para otimizar a cobertura e reduzir custos de instalação.
- **Planeamento da Produção:** Otimização de cronogramas de produção para maximizar a eficiência e reduzir tempos de inatividade.
- **Problema de Corte de Stock:** Otimização do corte de materiais para minimizar desperdícios.
- **Programação de Tarefas:** Planeamento e alocação de recursos em projetos complexos para garantir a conclusão eficiente.
- **Problema do Caminho Crítico:** Identificação de atividades críticas em projetos para evitar atrasos e melhorar a gestão de cronogramas.
- **Desenho de Redes de Distribuição de Energia:** Planeamento da rede elétrica para minimizar perdas e garantir a confiabilidade.
- **Otimização de Centrais Elétricas:** Planeamento da operação de centrais para maximizar a produção e minimizar custos operacionais.
- **Planeamento de Turnos de Enfermagem:** Otimização de horários de trabalho para garantir cobertura adequada e evitar sobrecarga de trabalho.
- **Distribuição de Medicamentos:** Planeamento logístico para garantir disponibilidade e minimizar custos de transporte.
- **Problemas de Alocação de Capital:** Distribuição eficiente de recursos financeiros entre projetos ou unidades de negócio.

3.4.8 Análise de Resultados

Instance	Mean	median	Min (best)	Max (worst)	STD Deviation	Number of Comparisons	Optimal Solution
capa	*1.8349e+07*	*1.8349e+07*	*1.8349e+07*	*1.8349e+07*	*1.39695e+07*	*980779800*	*17156454.478*
capb	*1.31144e+07*	*1.31144e+07*	*1.31144+07*	*1.31144e+07*	*2.803382e+07*	*950449500*	*12979071.582*
capc	*1.16478e+07*	*1.16478e+07*	*1.16478e+07*	*1.16478e+07*	*3.154382e+07*	*930229300*	*11505594.329*
mr1	*2349.86*	*2349.86*	*2349.86*	*2349.86*	*13.6352*	*62248498500*	*2349.856*
mr2	*2344.76*	*2344.76*	*2344.76*	*2344.76*	*5.2352*	*62248498500*	*2344.757*

mr3	*2183.23*	*2183.23*	*2183.23*	*2183.23*	*2.5168*	*62122998000*	*2183.235*
mr4	*2433.11*	*2433.11*	*2433.11*	*2433.11*	*1.8126*	*62248498500*	*2433.110*
mr5	*2344.35*	*2344.35*	*2344.35*	*2344.35*	*12.712*	*62248498500*	*2344.353*

Tabela 4: Resultados do Algoritmo Hill Climbing em diferentes Instâncias do UFLP

A tabela 4 apresenta os resultados do algoritmo Hill Climbing aplicados a diversas instâncias do UFLP. As métricas avaliadas incluem a média, mediana, valor mínimo (melhor), valor máximo (pior) e o desvio padrão das soluções obtidas para cada instância. As instâncias são rotuladas como *capa, capb, capc, mr1, mr2, mr3, mr4 e mr5*.

Análise das Instâncias “capa”, “capb” e “capc”:

capa

A instância "capa" apresenta uma média de 1.8349e+07, que é relativamente alta entre todas as instâncias. O desvio padrão (1.39695e+07) indica uma grande variabilidade nas soluções, sugerindo que o algoritmo explorou uma ampla gama de possíveis soluções. O número de comparações é bastante alto (980779800), e a solução ótima é 17156454.478, indicando que a qualidade da solução está próxima da média das soluções encontradas.

capb

A instância "capb" tem uma média de 1.31144e+07, que é menor que "capa" mas ainda significativa. O desvio padrão é ainda maior (2.803382e+07), sugerindo uma variabilidade ainda maior nas soluções. O número de comparações é ligeiramente menor que "capa" (950449500), e a solução ótima é 12979071.582, o que novamente está próximo da média das soluções.

capc

A instância "capc" apresenta a menor média entre "capa", "capb" e "capc" com um valor de 1.16478e+07. O desvio padrão é o maior entre as três (3.154382e+07), indicando a maior variabilidade. O número de comparações é 930229300, o menor entre as três instâncias, e a solução ótima é 11505594.329, que está alinhada com a média das soluções encontradas.

Análise das Instâncias “mr1”, “mr2”, “mr3”, “mr4” e “mr5”

mr1

A instância "mr1" apresenta uma média de 2349.86, que é a mais alta entre as instâncias "mr". O desvio padrão é 13.6352, indicando uma variabilidade relativamente baixa. O número de comparações é extremamente alto (62248498500), sugerindo uma análise muito detalhada. A solução ótima é 2349.856, muito próxima da média, indicando uma alta consistência nas soluções encontradas.

mr2

A instância "mr2" tem uma média de 2344.76, ligeiramente menor que "mr1". O desvio padrão é 5.2352, indicando ainda menor variabilidade. O número de comparações é o mesmo que "mr1" (62248498500), e a solução ótima é 2344.757, novamente muito próxima da média.

mr3

A instância "mr3" apresenta uma média de 2183.23, a menor entre as instâncias "mr". O desvio padrão é o menor (2.5168), indicando a menor variabilidade. O número de comparações é ligeiramente menor (62122998000), e a solução ótima é 2183.235, muito próxima da média.

mr4

A instância "mr4" tem uma média de 2433.11, a mais alta entre todas as instâncias "mr". O desvio padrão é 1.8126, o menor entre todas as instâncias "mr", indicando alta consistência nas soluções. O número de comparações é igual a "mr1" e "mr2" (62248498500), e a solução ótima é 2433.110, exatamente igual à média.

mr5

A instância "mr5" apresenta uma média de 2344.35, que é similar a "mr2". O desvio padrão é 12.712, maior que "mr4" mas ainda baixo. O número de comparações é 62248498500, e a solução ótima é 2344.353, muito próxima da média.

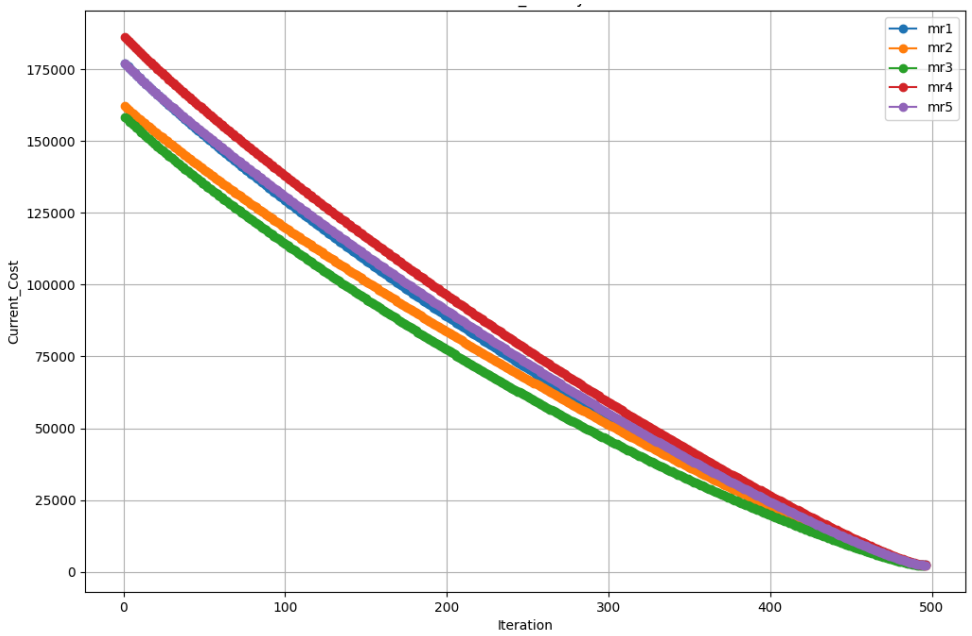


Figura 18: Iteration vs Current_Cost by MR Problem

A Figura 18 apresenta a evolução dos custos em função do número de iterações para as diferentes instâncias do problema MR (mr1, mr2, mr3, mr4, mr5). Cada linha colorida representa uma instância diferente do problema MR.

Observações Gerais

Tendência Geral: Todas as instâncias MR apresentam uma tendência decrescente no custo à medida que o número de iterações aumenta. Isso indica que o algoritmo está corretamente implementado, refinando as soluções e encontrando custos menores com mais iterações.

Convergência: Observa-se que todas as instâncias convergem para custos semelhantes por volta da 500ª iteração, indicando a estabilidade do algoritmo em encontrar soluções ótimas ou próximas do ótimo.

Variabilidade Inicial: Nos estágios iniciais, há uma maior variabilidade nos custos entre as diferentes instâncias. Isso pode ser atribuído às características específicas de cada instância e à fase exploratória inicial do algoritmo.

Análise por Instância

mr1: A instância mr1 começa com um custo relativamente alto e apresenta uma queda acentuada nas primeiras iterações, seguindo depois uma descida mais gradual até convergir com as outras instâncias.

mr2: Similar à mr1, mas com um custo inicial um pouco mais baixo. A tendência de queda é consistente e a convergência ocorre de forma estável.

mr3: Esta instância inicia com um custo ainda menor e apresenta uma redução constante no custo até convergir com as outras instâncias.

mr4: Começa com um custo elevado, mas rapidamente ajusta para se alinhar com as demais instâncias. A redução é gradual e estável.

mr5: A instância mr5 apresenta o menor custo inicial entre todas, com uma redução consistente e uma convergência final similar às demais.

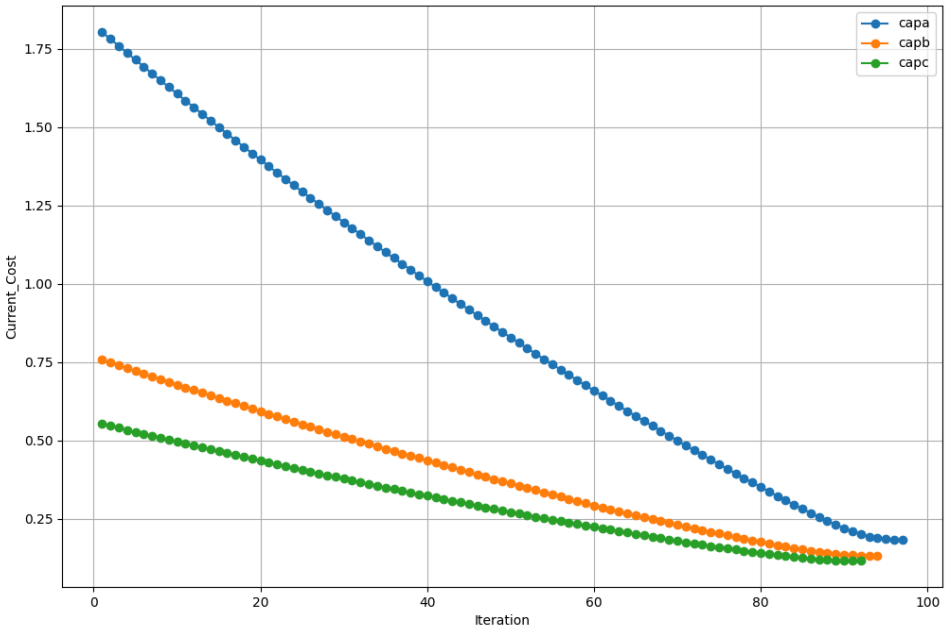


Figura 19: Iteration vs Current_Cost by CAP Problem

A Figura 19 apresenta a evolução dos custos em função do número de iterações para as diferentes instâncias do problema CAP (capa, capb, capc). Cada linha colorida representa uma instância diferente do problema CAP.

Observações Gerais

Tendência Geral: Todas as instâncias CAP apresentam uma tendência decrescente no custo à medida que o número de iterações aumenta. Isso indica que o algoritmo está corretamente implementado, refinando as soluções e encontrando custos menores com mais iterações.

Convergência: Observa-se que todas as instâncias convergem para custos próximos de 1, indicando a estabilidade do algoritmo em encontrar soluções ótimas ou próximas do ótimo.

Variabilidade Inicial: Nos estágios iniciais (primeiras 20 iterações), há uma maior variabilidade nos custos entre as diferentes instâncias. Isso pode ser atribuído às características específicas de cada instância e à fase exploratória inicial do algoritmo.

Análise por Instância

capa: A instância capa começa com o custo mais alto e apresenta uma queda acentuada nas primeiras 20 iterações, seguida de uma diminuição gradual até se estabilizar próximo de 1. Isso indica uma eficiência razoável do algoritmo em encontrar boas soluções desde o início.

capb: Apresenta um comportamento semelhante ao capa, começando com um custo moderado e descendo de forma consistente. A sua convergência final está bem alinhada com as outras instâncias, mostrando uma boa eficácia do algoritmo nesta instância.

capc: Esta instância começa com o custo mais baixo e tem uma descida mais suave e consistente. Chega a soluções de custo mais baixo relativamente rápido, mostrando uma boa eficácia do algoritmo nesta instância.

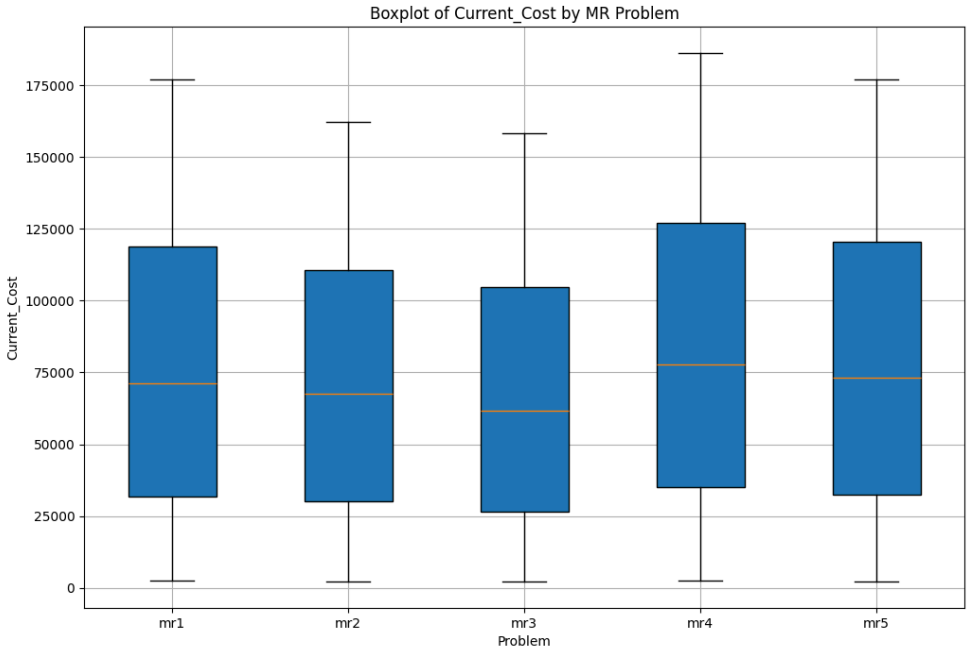


Figura 20: Boxplot of Current_Cost by MR Problem

A figura 20 apresenta um boxplot que mostra os custos atuais (Current_Cost) distribuídos por cinco diferentes tipos de problemas MR (mr1, mr2, mr3, mr4, mr5).

Observações Gerais

1. **Distribuição dos Custos:**
    - **mr1:** A mediana do custo para mr1 está em torno de 75,000. A caixa indica que a maioria dos valores está entre aproximadamente 50,000 e 125,000, com alguns valores extremos até cerca de 175,000.
    - **mr2:** A mediana do custo para mr2 é semelhante a mr1, em torno de 75,000. A maioria dos valores está entre aproximadamente 50,000 e 125,000, com menos variabilidade nos valores superiores.
    - **mr3:** A mediana do custo para mr3 é ligeiramente inferior, em torno de 65,000. Os valores estão concentrados entre aproximadamente 50,000 e 100,000, mostrando uma menor variabilidade comparada a mr1 e mr2.
    - **mr4:** A mediana do custo para mr4 está novamente em torno de 75,000, com uma maior dispersão de valores, chegando até 175,000.

    - **mr5:** A mediana do custo para mr5 é semelhante a mr4 e mr1, em torno de 75,000. Os valores estão concentrados entre aproximadamente 50,000 e 125,000.
2. **Variabilidade:**
    - **mr1, mr4 e mr5** apresentam maior variabilidade de custos, indicando uma maior dispersão nos custos.
    - **mr2 e mr3** têm uma menor variabilidade, indicando que os custos são mais consistentes e menos dispersos.

Análise por Problema

- **mr1:**
    - A mediana está em 75,000.
    - A variabilidade é alta, com um intervalo interquartil grande.
    - Existem valores extremos indicando alguns casos com custos significativamente mais altos.
- **mr2:**
    - A mediana está em 75,000.
    - A variabilidade é menor que mr1, com valores mais concentrados.
- **mr3:**
    - A mediana está em 65,000.
    - A variabilidade é menor, indicando maior consistência nos custos.
- **mr4:**
    - A mediana está em 75,000.
    - A variabilidade é alta, similar a mr1, com alguns valores extremos.
- **mr5:**
    - A mediana está em 75,000.
    - A variabilidade é alta, similar a mr1 e mr4.

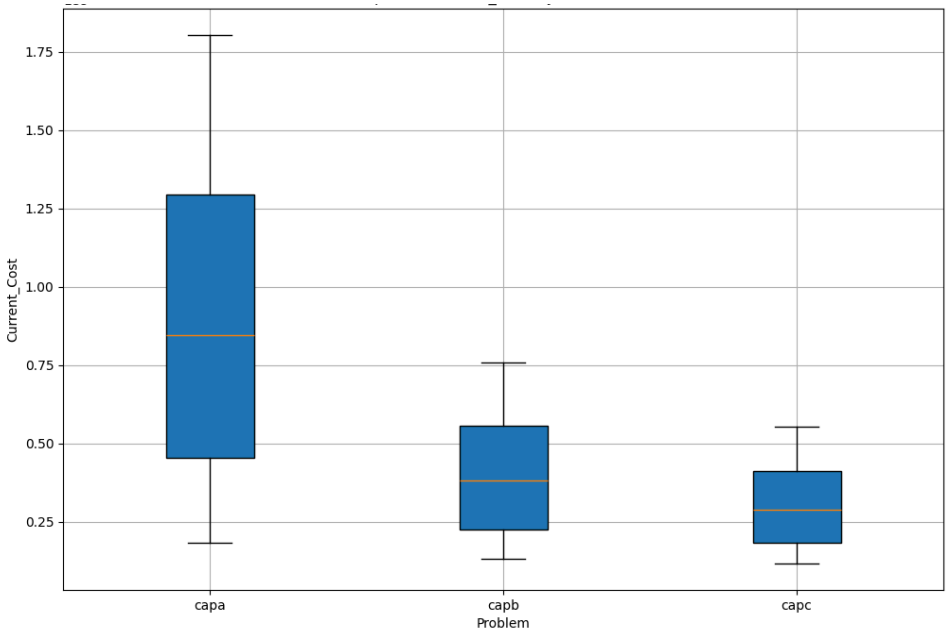


Figura 21: Boxplot of Current_Cost by CAP Problem

A Figura 21 é um boxplot que mostra os custos atuais (Current_Cost) distribuídos por três diferentes tipos de problemas CAP (capa, capb, capc).

Observações Gerais

1. **Distribuição dos Custos:**
    - **capa:** A mediana do custo para capa está em torno de 75 milhões. A caixa indica que a maioria dos valores está entre aproximadamente 50 milhões e 125 milhões, com alguns valores extremos até cerca de 175 milhões.
    - **capb:** A mediana do custo para capb é significativamente mais baixa, em torno de 25 milhões. A maioria dos valores está entre aproximadamente 12.5 milhões e 37.5 milhões.
    - **capc:** A mediana do custo para capc também está em torno de 25 milhões, semelhante a capb. Os valores estão concentrados entre aproximadamente 20 milhões e 35 milhões, mostrando a menor variabilidade entre os três problemas.
2. **Variabilidade:**
    - **capa** apresenta a maior variabilidade de custos, indicando que este problema tem uma maior dispersão nos custos.

- **capb e capc** têm uma menor variabilidade, indicando que os custos são mais consistentes e menos dispersos.

Análise por Problema

- **capa:**
  - A mediana está em 75 milhões.
  - A variabilidade é alta, com um intervalo interquartil grande.
  - Existem valores extremos indicando alguns casos muito mais caros.
- **capb:**
  - A mediana está em 25 milhões.
  - A variabilidade é menor que capa, com valores mais concentrados.
- **capc:**
  - A mediana está em 25 milhões.
  - A variabilidade é a menor entre os três, indicando maior consistência nos custos.

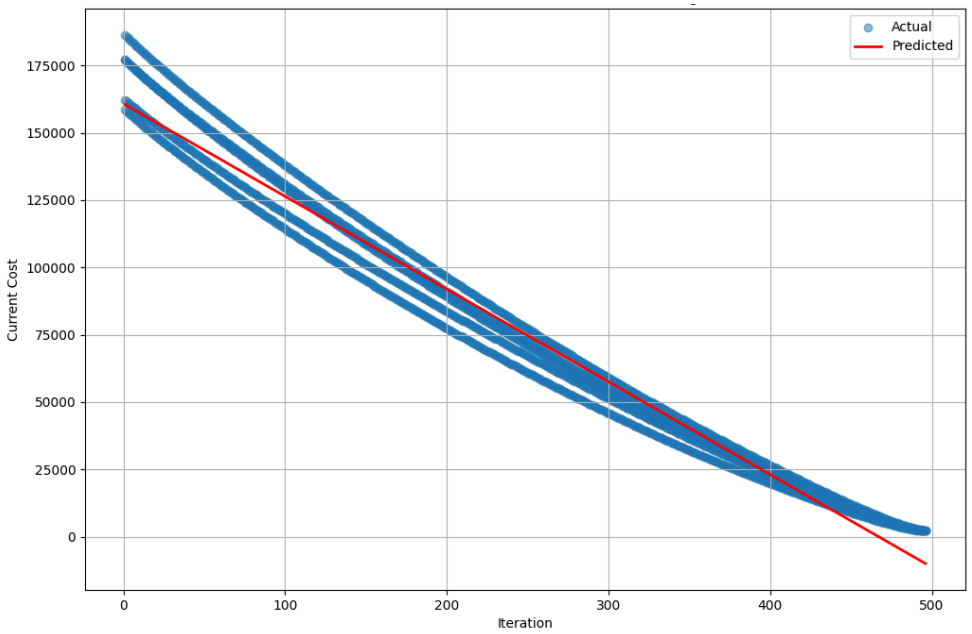


Figura 22: Iteration vs Current Cost (MR Problem) - Linear Regression

A figura 22 mostra um gráfico de dispersão com uma linha de regressão linear, comparando o custo atual (Current Cost) em função das iterações (Iteration) para um problema de Regressão Linear. Os pontos azuis representam os valores reais dos dados (Actual) e a linha vermelha representa os valores preditos pelo modelo de regressão (Predicted).

Observações Gerais:

- **Distribuição dos Dados**:
  - A maioria dos pontos de dados reais (azuis) está alinhada em faixas paralelas, indicando diferentes subgrupos ou clusters dentro do conjunto de dados.
  - A linha de regressão (vermelha) parece ser uma boa representação geral da tendência decrescente do custo ao longo das iterações.
- **Tendência**:
  - O custo atual diminui à medida que o número de iterações aumenta.
  - A linha de regressão linear prediz uma diminuição contínua do custo com o aumento das iterações.

Análise:

Com a análise do gráfico, é notável que, à medida que as iterações avançam, o custo corrente diminui de forma consistente, indicando uma melhoria no processo de otimização, o'que sugere que o algoritmo de optimização está a funcionar de maneira eficaz, ajustando progressivamente os parâmetros para minimizar os custos.

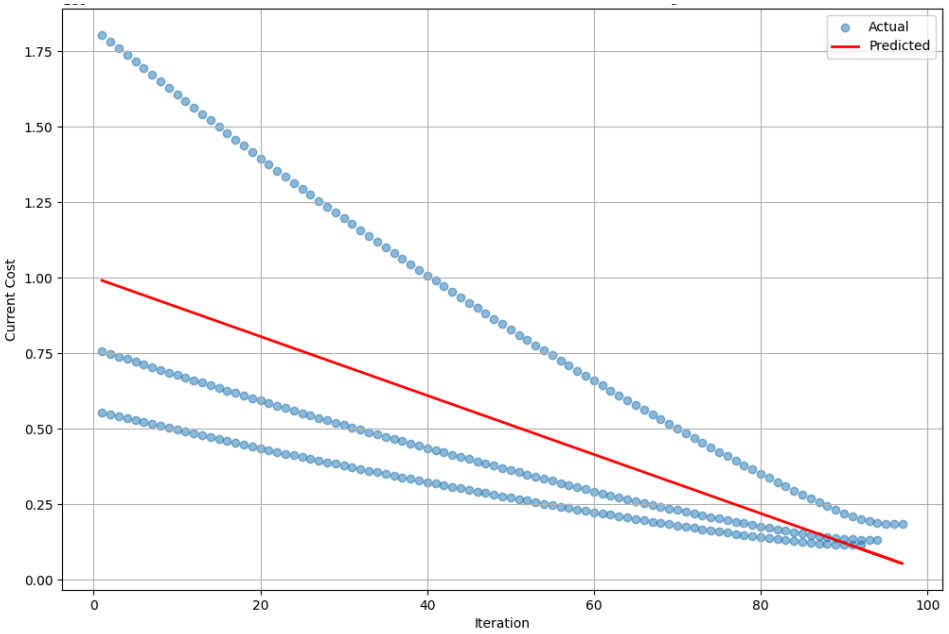


Figura 23: Iteration vs Current Cost (CAP Problem) - Linear Regression

A figura 6 mostra um gráfico de dispersão com uma linha de regressão linear, comparando o custo atual (Current Cost) em função das iterações (Iteration) para um problema CAP (Capacity Allocation Problem). Os pontos azuis representam os valores reais dos dados

(Actual) e a linha vermelha representa os valores preditos pelo modelo de regressão (Predicted).

Observações Gerais:

- **Distribuição dos Dados**:
    - A maioria dos pontos de dados reais (azuis) está alinhada em três faixas paralelas, indicando diferentes subgrupos ou clusters dentro do conjunto de dados.
    - A linha de regressão (vermelha) parece ser uma representação geral da tendência decrescente do custo ao longo das iterações, mas não captura todas as nuances dos dados.
- **Tendência**:
    - O custo atual diminui à medida que o número de iterações aumenta.
    - A linha de regressão linear prediz uma diminuição contínua do custo com o aumento das iterações.

Análise:

Com a análise do gráfico, é notável que, à medida que as iterações avançam, o custo corrente diminui de forma consistente, indicando uma melhoria no processo de otimização, o'que sugere que o algoritmo de optimização está a funcionar de maneira eficaz, ajustando progressivamente os parâmetros para minimizar os custos.

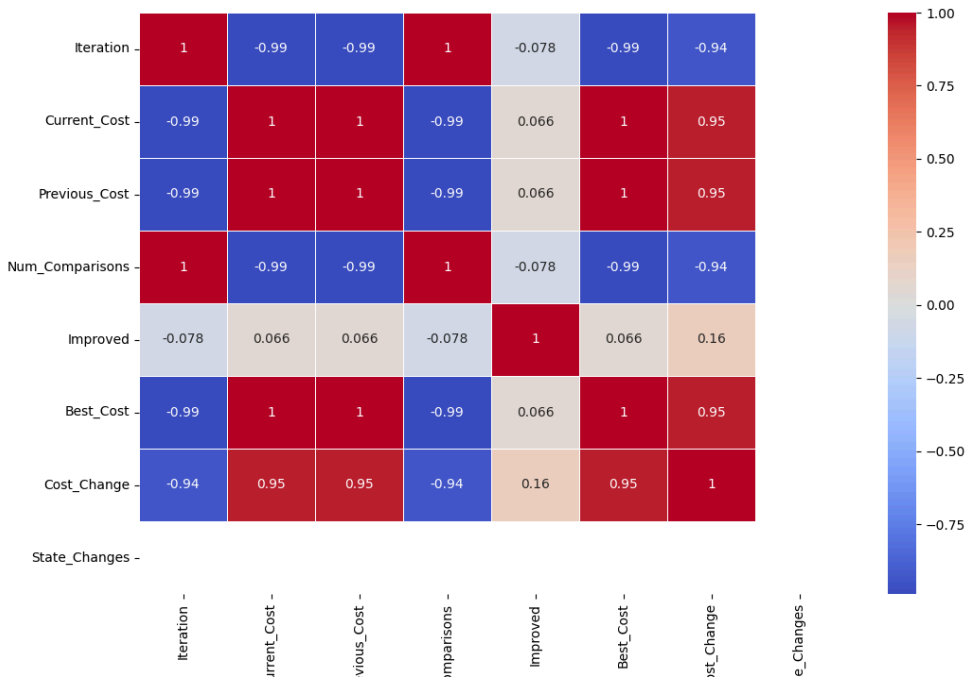


Figura 24: Correlation Matrix for MR Problems

A figura 24 é uma matriz correlação que mostra a relação entre várias variáveis associadas a problemas de Regressão (MR - Multiple Regression). As variáveis incluídas na matriz são: Iteration, Current_Cost, Previous_Cost, Num_Comparisons, Improved, Best_Cost, Cost_Change, e State_Changes. A matriz utiliza uma escala de cores para representar os coeficientes de correlação, variando de -1 (correlação negativa perfeita) a 1 (correlação positiva perfeita).

Observações Gerais:

- **Interpretação das Cores**:
  - Cores vermelhas intensas indicam uma forte correlação positiva.
  - Cores azuis intensas indicam uma forte correlação negativa.
  - Cores neutras ou claras indicam pouca ou nenhuma correlação.

Análise Detalhada das Correlações:

1. **Iteration**:
   - **Current_Cost**: Correlação negativa muito forte (-0.99), indicando que à medida que as iterações aumentam, o custo atual diminui significativamente.
   - **Previous_Cost**: Correlação negativa muito forte (-0.99), semelhante ao Current_Cost.

    - **Num_Comparisons**: Correlação positiva perfeita (1), sugerindo que o número de comparações aumenta linearmente com as iterações.
    - **Best_Cost**: Correlação negativa muito forte (-0.99), indicando que à medida que as iterações aumentam, o melhor custo encontrado também diminui.
    - **Cost_Change**: Correlação negativa muito forte (-0.94), sugerindo que as mudanças no custo também diminuem com o aumento das iterações.
2. **Current_Cost**:
    - **Previous_Cost**: Correlação positiva perfeita (1), indicando que os custos atuais e anteriores são praticamente idênticos.
    - **Num_Comparisons**: Correlação negativa muito forte (-0.99), indicando que um maior número de comparações está associado a um custo atual menor.
    - **Improved**: Correlação ligeiramente positiva (0.066), sugerindo uma relação muito fraca.
    - **Best_Cost**: Correlação positiva perfeita (1), indicando que o custo atual está fortemente relacionado com o melhor custo encontrado.
    - **Cost_Change**: Correlação positiva muito forte (0.95), indicando que as mudanças no custo estão fortemente relacionadas com o custo atual.
3. **Previous_Cost**:
    - Tem correlações idênticas ao Current_Cost com as outras variáveis, devido à correlação perfeita entre Current_Cost e Previous_Cost.
4. **Num_Comparisons**:
    - **Improved**: Correlação negativa muito fraca (-0.078), indicando pouca relação entre o número de comparações e melhorias.
    - **Best_Cost**: Correlação negativa muito forte (-0.99), sugerindo que mais comparações estão associadas a melhores custos.
    - **Cost_Change**: Correlação negativa muito forte (-0.94), indicando que mais comparações resultam em menores mudanças de custo.
5. **Improved**:
    - **Best_Cost**: Correlação positiva muito fraca (0.066), indicando pouca relação entre melhorias e o melhor custo.
    - **Cost_Change**: Correlação ligeiramente positiva (0.16), sugerindo que melhorias têm uma relação fraca com mudanças no custo.
6. **Best_Cost**:
    - **Cost_Change**: Correlação positiva muito forte (0.95), indicando que melhores custos estão fortemente relacionados com mudanças no custo.

Conclusões:

- **Correlações Fortes**:

    - As variáveis Iteration, Current_Cost, Previous_Cost, Num_Comparisons, Best_Cost e Cost_Change têm correlações muito fortes entre si, indicando que estas variáveis estão fortemente interligadas e influenciam-se mutuamente de maneira significativa.
- **Correlações Fracas**:
    - A variável Improved mostra correlações fracas com quase todas as outras variáveis, sugerindo que melhorias podem não estar diretamente relacionadas com as outras variáveis na matriz.
- **Interpretação Geral**:
    - A matriz de correlação revela relações muito fortes e previsíveis entre a maioria das variáveis, exceto para a variável Improved. As fortes correlações negativas com Iteration indicam que aumentar as iterações está consistentemente associado à redução dos custos.

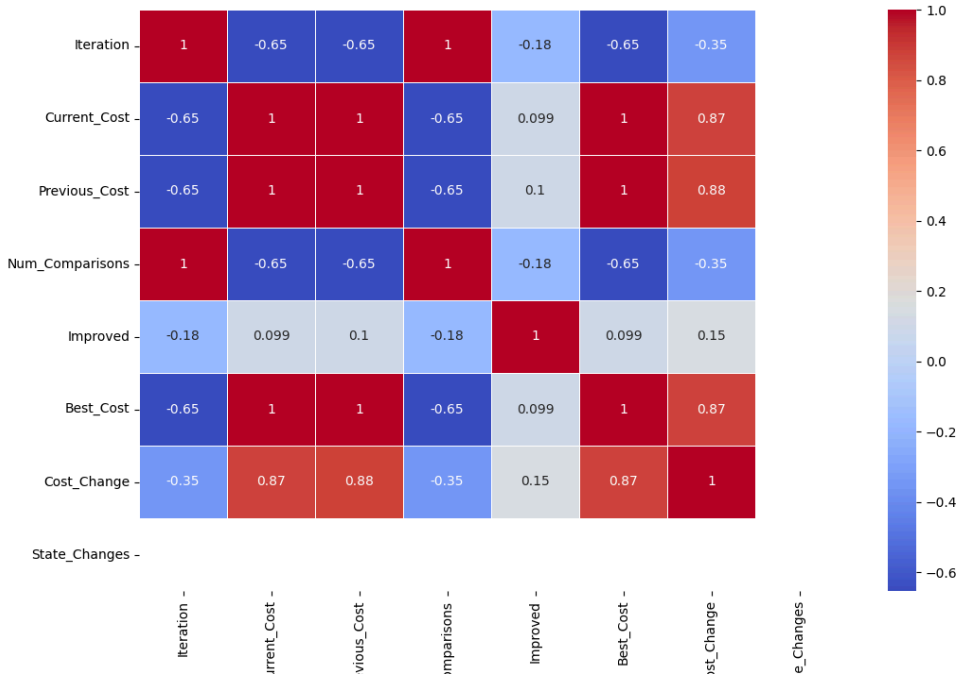


Figura 25: Correlation Matrix for CAP Problems

A figura 23 é uma matriz correlação que mostra a relação entre várias variáveis associadas a problemas de Regressão (CAP Problem) . As variáveis incluídas na matriz são: Iteration, Current_Cost, Previous_Cost, Num_Comparisons, Improved, Best_Cost, Cost_Change, e State_Changes. A matriz utiliza uma escala de cores para representar os coeficientes de correlação, variando de -1 (correlação negativa perfeita) a 1 (correlação positiva perfeita).

Observações Gerais:

- **Interpretação das Cores**:
    - **Cores Vermelhas Intensas**: Indicam uma forte correlação positiva.
    - **Cores Azuis Intensas**: Indicam uma forte correlação negativa.
    - **Cores Neutras ou Claras**: Indicam pouca ou nenhuma correlação.

Análise das Correlações:

- **Iteration**:
    - **Current_Cost**: Correlação negativa forte (-0.65), indicando que à medida que as iterações aumentam, o custo atual tende a diminuir.
    - **Previous_Cost**: Correlação negativa forte (-0.65), semelhante à correlação com Current_Cost.
    - **Num_Comparisons**: Correlação positiva perfeita (1), sugerindo que o número de comparações aumenta linearmente com as iterações.
    - **Improved**: Correlação negativa fraca (-0.18), indicando pouca relação entre o número de iterações e a melhoria.
    - **Best_Cost**: Correlação negativa forte (-0.65), indicando que à medida que as iterações aumentam, o melhor custo encontrado também diminui.
    - **Cost_Change**: Correlação negativa moderada (-0.35), sugerindo que as mudanças no custo também diminuem com o aumento das iterações.
- **Current_Cost**:
    - **Previous_Cost**: Correlação positiva perfeita (1), indicando que os custos atuais e anteriores são praticamente idênticos.
    - **Num_Comparisons**: Correlação negativa forte (-0.65), indicando que um maior número de comparações está associado a um custo atual menor.
    - **Improved**: Correlação ligeiramente positiva (0.099), sugerindo uma relação muito fraca.
    - **Best_Cost**: Correlação positiva perfeita (1), indicando que o custo atual está fortemente relacionado com o melhor custo encontrado.
    - **Cost_Change**: Correlação positiva forte (0.87), indicando que as mudanças no custo estão fortemente relacionadas com o custo atual.
- **Previous_Cost**:
    - Tem correlações idênticas ao Current_Cost com as outras variáveis, devido à correlação perfeita entre Current_Cost e Previous_Cost.
- **Num_Comparisons**:

    - **Improved**: Correlação negativa fraca (-0.18), indicando pouca relação entre o número de comparações e melhorias.
    - **Best_Cost**: Correlação negativa forte (-0.65), sugerindo que mais comparações estão associadas a melhores custos.
    - **Cost_Change**: Correlação negativa moderada (-0.35), indicando que mais comparações resultam em menores mudanças de custo.
- **Improved**:
    - **Best_Cost**: Correlação ligeiramente positiva (0.099), indicando pouca relação entre melhorias e o melhor custo.
    - **Cost_Change**: Correlação ligeiramente positiva (0.15), sugerindo que melhorias têm uma relação fraca com mudanças no custo.
- **Best_Cost**:
    - **Cost_Change**: Correlação positiva forte (0.87), indicando que melhores custos estão fortemente relacionados com mudanças no custo.

Conclusões:

- **Correlações Fortes**:
    - As variáveis **Iteration, Current_Cost, Previous_Cost, Num_Comparisons, Best_Cost** e **Cost_Change** têm correlações fortes entre si, indicando que estas variáveis estão fortemente interligadas e influenciam-se mutuamente de maneira significativa.
- **Correlações Fracas**:
    - A variável **Improved** mostra correlações fracas com quase todas as outras variáveis, sugerindo que melhorias podem não estar diretamente relacionadas com as outras variáveis na matriz.
- **Interpretação Geral**:
    - A matriz de correlação revela relações fortes e previsíveis entre a maioria das variáveis, exceto para a variável **Improved**. As fortes correlações negativas com **Iteration** indicam que aumentar as iterações está consistentemente associado à redução dos custos.

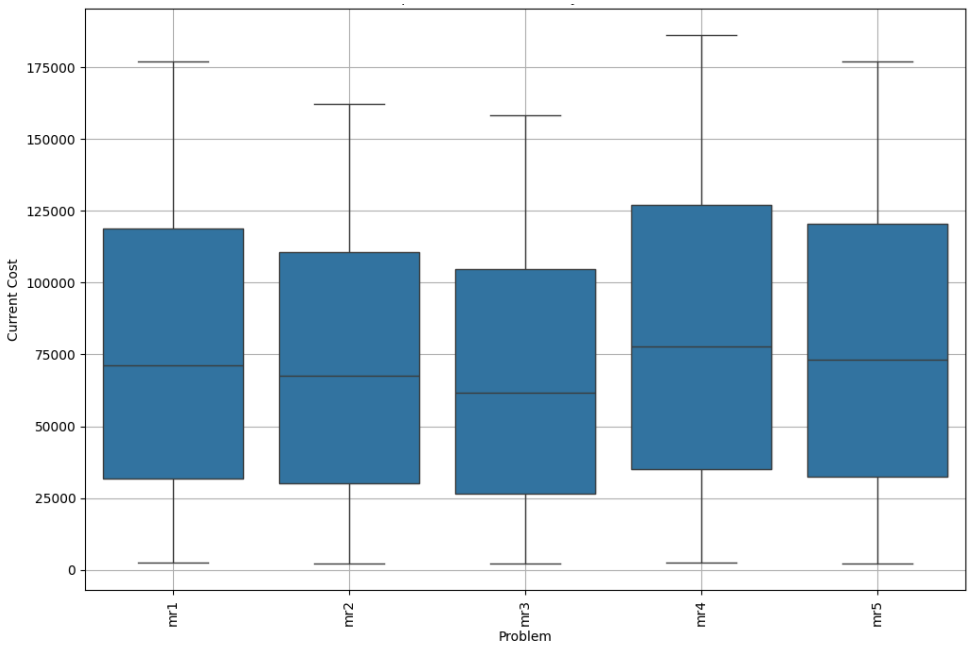


Figura 26: Boxplot of Current Cost by MR Problem

A figura 24 apresenta um boxplot que mostra os custos atuais (Current Cost) distribuídos por cinco diferentes tipos de problemas MR (mr1, mr2, mr3, mr4, mr5).

Observações Gerais

Distribuição dos Custos:

- **mr1:** A mediana do custo para mr1 está em torno de 75 mil. A caixa indica que a maioria dos valores está entre aproximadamente 50 mil e 125 mil, com alguns valores extremos até cerca de 175 mil.
- **mr2:** A mediana do custo para mr2 está em torno de 50 mil. A maioria dos valores está entre aproximadamente 25 mil e 100 mil.
- **mr3:** A mediana do custo para mr3 está em torno de 50 mil, semelhante a mr2. Os valores estão concentrados entre aproximadamente 25 mil e 100 mil.
- **mr4:** A mediana do custo para mr4 está em torno de 100 mil. A caixa indica que a maioria dos valores está entre aproximadamente 50 mil e 150 mil, com alguns valores extremos até cerca de 175 mil.
- **mr5:** A mediana do custo para mr5 está em torno de 75 mil. A maioria dos valores está entre aproximadamente 50 mil e 125 mil, com alguns valores extremos até cerca de 175 mil.

Variabilidade:

- **mr1** apresenta uma variabilidade de custos moderada, indicando que o problema têm uma dispersão razoável nos custos.
- **mr2** e **mr3** têm uma variabilidade menor, indicando que os custos são mais consistentes e menos dispersos.
- **mr4** apresenta a maior variabilidade de custos, indicando uma dispersão significativa nos valores.
- **mr5** tem uma variabilidade de custos similar a mr1.

Análise por Problema

- **mr1:**
  - A mediana está em 75 mil.
  - A variabilidade é moderada, com um intervalo interquartil que vai de aproximadamente 50 mil a 125 mil.
  - Existem valores extremos indicando alguns casos mais caros.
- **mr2:**
  - A mediana está em 50 mil.
  - A variabilidade é menor, com valores mais concentrados entre aproximadamente 25 mil e 100 mil.
- **mr3:**
  - A mediana está em 50 mil.
  - A variabilidade é semelhante a mr2, com valores concentrados entre aproximadamente 25 mil e 100 mil.
- **mr4:**
  - A mediana está em 100 mil.
  - A variabilidade é alta, com um intervalo interquartil grande e alguns valores extremos.
- **mr5:**
  - A mediana está em 75 mil.
  - A variabilidade é moderada, com um intervalo interquartil que vai de aproximadamente 50 mil a 125 mil.
  - Existem valores extremos indicando alguns casos mais caros.

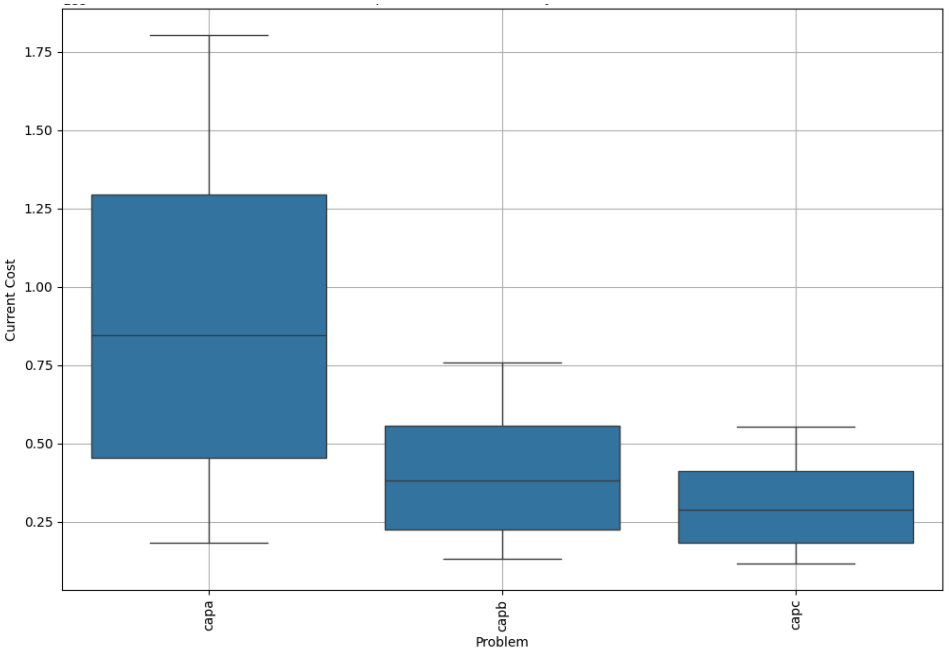


Figura 27: Boxplot of Current Cost by CAP Problem

A figura 25 apresenta um boxplot que mostra os custos atuais (Current Cost) distribuídos por três diferentes tipos de problemas CAP (capa, capb, capc).

Observações Gerais

Distribuição dos Custos:

- **capa:** A mediana do custo para capa está em torno de 75 milhões. A caixa indica que a maioria dos valores está entre aproximadamente 50 milhões e 125 milhões, com alguns valores extremos até cerca de 175 milhões.
- **capb:** A mediana do custo para capb é significativamente mais baixa, em torno de 25 milhões. A maioria dos valores está entre aproximadamente 12.5 milhões e 37.5 milhões.
- **capc:** A mediana do custo para capc também está em torno de 25 milhões, semelhante a capb. Os valores estão concentrados entre aproximadamente 20 milhões e 35 milhões, mostrando a menor variabilidade entre os três problemas.

Variabilidade:

- **capa** apresenta a maior variabilidade de custos, indicando que o problema têm uma maior dispersão nos custos.

- **capb** e **capc** têm uma menor variabilidade, indicando que os custos são mais consistentes e menos dispersos.

Análise por Problema

- **capa:**
  - A mediana está em 75 milhões.
  - A variabilidade é alta, com um intervalo interquartil grande.
  - Existem valores extremos indicando alguns casos muito mais caros.
- **capb:**
  - A mediana está em 25 milhões.
  - A variabilidade é menor que capa, com valores mais concentrados.
- **capc:**
  - A mediana está em 25 milhões.
  - A variabilidade é a menor entre os três, indicando maior consistência nos custos.

4. Conclusão da Análise dos algoritmos implementados

De maneira geral, ficamos satisfeitos com os resultados obtidos através da implementação dos diferentes algoritmos. Cada algoritmo apresentou características únicas, e vantagens diversas.

O Binary Crow Search apresentou ótimos resultados em um tempo de execução eficiente. Este algoritmo destacou-se pela sua exploração eficaz, o que acabou por resultar em bons resultados.

O GRASP também teve um bom desempenho, apesar de encontrarmos alguns problemas relacionados ao balanço em alfa, que é responsável por controlar a exploração. Em alguns casos, levou o algoritmo a entrar num estado de “sobre-exploração”, o que acabou por afetar o tempo de execução do mesmo, embora a qualidade do mesmo se tenha mantido boa.

Embora o Simulated Annealing não tenha produzido resultados extremamente bons, mostrou-se interessante devido à capacidade do mesmo de incorporar os conceitos dos outros algoritmos, como o Tabu Search e Genético. Integramos alguns mecanismos de mutação e crossover, típicos dos algoritmos referidos anteriormente, o que proporcionou uma grande diversidade das soluções. Este hibridismo mostrou-se promissor.

O Hill Climbing, apesar de ser uma abordagem mais simples, revelou-se interessante por diferentes motivos. Embora o tempo de execução tenha sido mais longo, o mesmo permitiu a recolha de uma quantidade maior de “logs”, o que proporcionou uma análise mais aprofundada dos resultados. A simplicidade do algoritmo, direta e iterativa ofereceu uma perspectiva diferente dos outros algoritmos implementados.

Cada algoritmo implementado trouxe contribuições significativas e algum aprendizado valioso. Cada uma destas abordagens ampliou o nosso entendimento sobre as diferentes técnicas possíveis e as aplicações práticas.

5. Conclusões e Trabalho Futuro

Em conclusão, o projeto de Análise Algorítmica e Otimização representou uma valiosa jornada de aprendizado e descoberta. A escolha de C++ como linguagem principal, juntamente com o uso de CMake, não apenas expandiu as nossas habilidades técnicas, mas também proporcionou o desenvolvimento de boas práticas de desenvolvimento. A implementação de testes unitários com thresholds demonstrou um compromisso com a qualidade das soluções, garantindo que os algoritmos atendessem a critérios mínimos de desempenho.

A utilização de estatísticas com Python e Pandas foi crucial para a análise do comportamento dos algoritmos, oferecendo insights valiosos sobre o seu desempenho e eficácia em diferentes cenários. Este aspecto do projeto não apenas validou as implementações, mas também nos permitiu comparar e compreender os algoritmos em um nível mais profundo, explorando até certo nível conceitos de *state-of-the-art*.

No entanto, o projeto não esteve isento de desafios. A complexidade de algumas implementações em C++ exigiu uma cuidadosa gestão de tempo, e a integração entre diferentes ferramentas e linguagens nem sempre foi trivial. Além disso, identificar e resolver problemas de desempenho e otimização foi uma tarefa contínua, destacando a importância da análise crítica e refinamento iterativo.

Em resumo, este projeto foi uma experiência enriquecedora que não apenas ampliou o nosso conhecimento técnico, mas também nos preparou melhor para desafios futuros na área de desenvolvimento de software.

5. Referências Bibliográficas

1. Powerpoints disponibilizados na plataforma moodle.
2. Artigos disponiblizados na plataforma moddle
3. Operations Research - An introduction Tenth Edition Global Edition - Hamdy A. Taha
4. Introduction to Operations Research, 11th Edition - Frederick Lieberman McGraw Hill

6. Referências WWW

1. https://github.com/memento/GeneticAlgorithm/tree/master
2. https://gist.github.com/jasantillan/8252b97e8c5413040c77ab7bd742a6b0
3. https://www.learncpp.com/
4. https://dabeenl.github.io/IE631_lecture20_note.pdf
5. https://proceedings.mlr.press/v9/lazic10a/lazic10a.pdf
6. https://ugtcs.berkeley.edu/src/approx-sp19/scribe-notes-6.pdf
7. https://adam-rumpf.github.io/documents/cplex_in_cpp.pdf
8. https://en.wikipedia.org/wiki/Simulated_annealing
9. https://math.mit.edu/~rmd/46512/simanneal.pdf
10. https://enac.hal.science/hal-01887543/document
11. https://www.researchgate.net/publication/351894160_Binary_crow_search_algorithm_for_the_uncapacitated_facility_location_problem
12. https://www.researchgate.net/publication/339663624_Solving_practical_economic_load_dispatch_problem_using_crow_search_algorithm
13. https://www.researchgate.net/publication/322282157_A_GRASP_algorithm_based_new_heuristic_for_the_capacitated_location_routing_problem
14. https://link.springer.com/content/pdf/10.1007/s10479-007-0193-1.pdf
15. https://www.researchgate.net/publication/264911477_A_GRASP_algorithm_for_the_single_source_uncapacitated_minimum_concave-cost_network_flow_problem
16. https://link.springer.com/content/pdf/10.1007/s10288-018-0397-z.pdf
17. https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/
18. https://pages.cs.wisc.edu/~jerryzhu/cs540/handouts/hillclimbing.pdf
19. https://www.parthapakray.com/ai/2023/Lecture%20on%20Hill%20Climbing%20Algorithm.pdf
20. https://link.springer.com/content/pdf/10.1007/s00521-016-2328-2.pdf
21. https://people.orie.cornell.edu/shmoys/pdf/approx00.pdf
22. https://cse.buffalo.edu/~shil/papers/UFL-ICALP2011.pdf
23. https://webdocs.cs.ualberta.ca/~mreza/courses/Approx15/week4.pdf
24. https://www.geeksforgeeks.org/what-is-tabu-search/
25. https://www.youtube.com/watch?v=i1IJAVhCn6U
26. https://github.com/HenrywlChuang/Meta-heuristic-Algorithm/blob/main/src/TS.cpp#L72
27. https://www.researchgate.net/publication/226263988_A_Tabu_Search_Heuristic_for_the_Uncapacitated_Facility_Location_Problem
28. https://run.unl.pt/bitstream/10362/139556/1/Mendes_2022.pdf

29. http://wexler.free.fr/library/files/kirkpatrick%20(1983)%20optimization%20by%20simulated%20annealing.pdf
30. https://link.springer.com/article/10.1007/s00521-021-06107-2
31. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6105025/

7. Anexos

Algorithm.hpp

```
#pragma once
#include "../problem/Problem.hpp"

/**
```

```
 * @brief Namespace for the algorithm classes
 */
namespace algorithm {
	/**
	 * @brief Abstract class for the algorithms
	 */
	class Algorithm {
	public:
		virtual std::vector<std::pair<int, int>> solve(const Problem&
problem) const = 0;
		virtual ~Algorithm() {}
	};
}
```

BinaryCrowSearch.hpp

```
#pragma once

#include "Algorithm.hpp"
#include <vector>

namespace algorithm {

    class CrowSearchAlgorithm : public Algorithm {
    private:
        static double closed_interval_rand(double x0, double x1);
        static double UFLP(int loc, int cus, const
std::vector<std::vector<double>>& customer,
                           const std::vector<double>& location, const
std::vector<bool>& per);

        int population_size;
        double awareness_probability;
        int function_evaluations;

    public:
        CrowSearchAlgorithm(int pop_size, double ap, int func_evals)
            : population_size(pop_size), awareness_probability(ap),
function_evaluations(func_evals) {}

        std::vector<std::pair<int, int>> solve(const Problem& problem)
const override;
    };
```

```
} // namespace algorithm
```

BinaryCrowSearch.cpp

```
    #include "CrowSearchAlgorithm.hpp"
    #include <random>
    #include <algorithm>
    #include <ctime>
    #include <cmath>
    #include <cfloat>
    #include <vector>
    #include <iostream>

    namespace algorithm {

        std::default_random_engine
generator(static_cast<unsigned>(std::time(nullptr)));

        double CrowSearchAlgorithm::closed_interval_rand(double x0, double
x1) {
            std::uniform_real_distribution<double> distribution(x0, x1);
            return distribution(generator);
        }

        double CrowSearchAlgorithm::UFLP(int loc, int cus, const
std::vector<std::vector<double>>& customer,
            const std::vector<double>& location, const std::vector<bool>&
per)
        {
            double cost = 0;
            double min_cost = 0;
            for (int i = 0; i < loc; ++i)
                if (per[i])
                    cost += location[i];

            for (int i = 0; i < cus; ++i) {
                min_cost = DBL_MAX;
                for (int j = 0; j < loc; ++j) {
                    if (per[j] && customer[i][j] < min_cost)
                        min_cost = customer[i][j];
                }
                cost += min_cost;
            }
```

```
            return cost;
        }

        std::vector<std::pair<int, int>> CrowSearchAlgorithm::solve(const
Problem& problem) const {
            int loc = problem.getNumberOfWarehouses();
            int cus = problem.getNumberOfCustomers();

            const auto& customers = problem.getCustomers();
            const auto& warehouses = problem.getWarehouses();

            std::vector<double> location(loc);
            std::vector<std::vector<double>> customer(cus,
std::vector<double>(loc));

            for (int i = 0; i < loc; ++i)
                location[i] = warehouses[i].getFixedCost();

            for (int i = 0; i < cus; ++i) {
                const std::vector<double>& costs =
customers[i].getAllocationCosts();
                for (int j = 0; j < loc; ++j) {
                    customer[i][j] = costs[j];
                }
            }

            const int N = population_size;  // Population size
            const double AP = awareness_probability;  // Awareness
probability
            const int MAX_ITER = function_evaluations / N;  // Number of
iterations

            std::vector<bool> x(loc);
            std::vector<int> follow(N);

            std::vector<double> obj_crows(N, 0);
            std::vector<double> obj_memory(N, 0);

            std::vector<std::vector<bool>> x_crows(N,
std::vector<bool>(loc));
            std::vector<std::vector<bool>> x_memory(N,
std::vector<bool>(loc));

            double global_best = DBL_MAX;

            std::vector<std::pair<int, int>> final_assignments;

            // Memory initialization with more strategic approach
```

```
            for (int i = 0; i < N; i++) {
                for (int j = 0; j < loc; j++) {
                    if (closed_interval_rand(0, 1) < 0.5) {
                        x_crows[i][j] = false;
                        x_memory[i][j] = false;
                        continue;
                    }
                    x_crows[i][j] = true;
                    x_memory[i][j] = true;
                }
            }

            for (int i = 0; i < N; i++) {
                obj_memory[i] = UFLP(loc, cus, customer, location,
x_crows[i]);
            }

            // Iterations start
            for (int iter = 0; iter < MAX_ITER; iter++) {
                for (int i = 0; i < N; i++) {
                    obj_crows[i] = UFLP(loc, cus, customer, location,
x_crows[i]);

                    if (obj_crows[i] < obj_memory[i]) {
                        obj_memory[i] = obj_crows[i];

                        for (int j = 0; j < loc; j++) {
                            x_memory[i][j] = x_crows[i][j];
                        }
                    }

                    if (obj_memory[i] < global_best) {
                        global_best = obj_memory[i];
                        final_assignments.clear();

                        // Update final_assignments with the new best
assignments
                        for (int k = 0; k < cus; ++k) {
                            int warehouse_index = -1;
                            double min_cost = DBL_MAX;
                            for (int j = 0; j < loc; ++j) {
                                if (x_memory[i][j] && customer[k][j] <
min_cost) {
                                    min_cost = customer[k][j];
                                    warehouse_index = j;
                                }
                            }
                            final_assignments.emplace_back(k,
```

```
warehouse_index);
                        }
                    }
                }

                for (int i = 0; i < N; ++i) {
                    follow[i] = std::ceil(N * closed_interval_rand(0, 1)) -
1;
                }

                for (int i = 0; i < N; ++i) {
                    if (closed_interval_rand(0, 1) > AP) {
                        for (int j = 0; j < loc; ++j) {
                            x_crows[i][j] = x_memory[i][j] ^ ((std::rand()
& 1) & (x_memory[follow[i]][j] ^ x_memory[i][j]));
                        }
                    }
                    else {
                        for (int j = 0; j < loc; ++j) {
                            x_crows[i][j] = closed_interval_rand(0, 1) <
0.5;
                        }
                    }
                }
            }

            return final_assignments;
        }

    } // namespace algorithm
```

GRASP.hpp

```
#pragma once
#include "../problem/Problem.hpp"
#include <vector>
#include <limits>

namespace algorithm {

    class GRASP {
    private:
        std::vector<std::vector<double>> allocation_costs;
        std::vector<double> fixed_costs;
        double current_objective;
```

```
        std::vector<int> best_assignment;
        int number_of_customers;
        const double MAX_DOUBLE = std::numeric_limits<double>::max();
        int number_of_warehouses;
        std::vector<bool> warehouse_open;
        std::vector<int> customer_assignment;
        std::vector<bool> best_warehouse_open;
        double alpha;

        bool CloseWarehouse();
        bool OpenWarehouse();
        bool OpenCloseWarehouse();
        double LocalSearchHeuristic(double objective);
        double GreedyRandomizedConstructive();
        double ReassignCustomers();

    public:
        GRASP(double alpha);
        void initialize(const Problem& problem);
        std::vector<std::pair<int, int>> solve(const Problem& problem);
    };
}
```

GRASP.cpp

```
#include "GRASP.hpp"
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <unordered_map>
#include <algorithm>
#include <thread>
#include <vector>

namespace algorithm
{
    /**
     * @brief Constructor for the GRASP class.
     *
     * @param num_iterations Number of iterations for the GRASP algorithm.
     * @param num_iterations_local Number of iterations for the local
search heuristic.
     * @param alpha Parameter for the greedy randomized constructive phase.
     */
    GRASP::GRASP(double alpha)
```

```
        : alpha(alpha) {}

    /**
     * @brief Initializes the GRASP algorithm with the given problem.
     *
     * @param problem The problem instance to solve.
     *
     * This function initializes various parameters and data structures
required for the GRASP algorithm,
     * including the number of customers, the number of warehouses, fixed
costs, allocation costs, and assignment vectors.
     */
    void GRASP::initialize(const Problem &problem){
        number_of_customers = problem.getNumberOfCustomers();
        number_of_warehouses = problem.getNumberOfWarehouses();

        fixed_costs.resize(number_of_warehouses);
        allocation_costs.resize(number_of_warehouses,
std::vector<double>(number_of_customers));

        for (int i = 0; i < number_of_warehouses; ++i){
            fixed_costs[i] = problem.getWarehouses()[i].getFixedCost();
        }

        for (int j = 0; j < number_of_customers; ++j){
            for (int i = 0; i < number_of_warehouses; ++i){
                allocation_costs[i][j] =
problem.getCustomers()[j].getAllocationCosts()[i];
            }
        }

        best_assignment.resize(number_of_customers);
        warehouse_open.resize(number_of_warehouses, false);
        customer_assignment.resize(number_of_customers);
    }

    /**
     * @brief Reassigns customers to the currently open warehouses to
minimize costs.
     *
     * @return The total cost of the current assignment.
     *
     * This function calculates the minimum cost assignment of customers to
open warehouses.
     * It updates the customer assignment and calculates the total cost
considering both fixed and allocation costs.
     */
    double GRASP::ReassignCustomers(){
```

```
        std::vector<double> customer_min_cost(number_of_customers,
MAX_DOUBLE);
        std::vector<int> best_assignment_temp(number_of_customers, -1);

        for (int j = 0; j < number_of_customers; ++j){
            double assign_cost = MAX_DOUBLE;
            int best_warehouse = -1;

            for (int i = 0; i < number_of_warehouses; ++i){
                if (warehouse_open[i] && allocation_costs[i][j] <
assign_cost){
                    assign_cost = allocation_costs[i][j];
                    best_warehouse = i;
                }
            }
            customer_min_cost[j] = assign_cost;
            best_assignment_temp[j] = best_warehouse;
        }

        double cost = 0;

        for (int i = 0; i < number_of_warehouses; ++i){
            if (warehouse_open[i]){
                cost += fixed_costs[i];
            }
        }

        for (int j = 0; j < number_of_customers; ++j){
            if (best_assignment_temp[j] != -1){
                cost += customer_min_cost[j];
                customer_assignment[j] = best_assignment_temp[j];
            }
        }

        return cost;
    }

    /**
     * @brief Attempts to close one of the currently open warehouses to
reduce costs.
     *
     * @return True if a warehouse was successfully closed to reduce costs,
false otherwise.
     *
     * This function iterates over all open warehouses and attempts to
close each one,
     * reassigning customers to minimize costs. If closing a warehouse
reduces the total cost,
```

```
     * the function keeps it closed.
     */
    bool GRASP::CloseWarehouse(){
        int best_warehouse = -1;

        for (int i = 0; i < number_of_warehouses; ++i){
            if (warehouse_open[i]){
                warehouse_open[i] = false;
                double cost = ReassignCustomers();

                if (cost < current_objective){
                    current_objective = cost;
                    best_warehouse = i;
                }

                warehouse_open[i] = true;
            }
        }

        if (best_warehouse != -1){
            warehouse_open[best_warehouse] = false;
            return true;
        }

        return false;
    }

    /**
     * @brief Attempts to open one of the currently closed warehouses to
reduce costs.
     *
     * @return True if a warehouse was successfully opened to reduce costs,
false otherwise.
     *
     * This function iterates over all closed warehouses and attempts to
open each one,
     * reassigning customers to minimize costs. If opening a warehouse
reduces the total cost,
     * the function keeps it open.
     */
    bool GRASP::OpenWarehouse(){
        int best_warehouse = -1;

        for (int i = 0; i < number_of_warehouses; ++i){
            if (!warehouse_open[i]){
                warehouse_open[i] = true;
                double cost = ReassignCustomers();
```

```
                if (cost < current_objective){
                    current_objective = cost;
                    best_warehouse = i;
                }

                warehouse_open[i] = false;
            }
        }

        if (best_warehouse != -1){
            warehouse_open[best_warehouse] = true;
            return true;
        }

        return false;
    }

    /**
     * @brief Attempts to simultaneously open one closed warehouse and
close one open warehouse to reduce costs.
     *
     * @return True if a pair of warehouses was successfully opened and
closed to reduce costs, false otherwise.
     *
     * This function iterates over all pairs of open and closed warehouses,
attempting to open a closed warehouse
     * and close an open warehouse simultaneously. If this operation
reduces the total cost, the function keeps the
     * changes.
     */
    bool GRASP::OpenCloseWarehouse(){
        int best_open = -1;
        int best_closed = -1;

        for (int i1 = 0; i1 < number_of_warehouses; ++i1){
            if (!warehouse_open[i1]){
                for (int i2 = 0; i2 < number_of_warehouses; ++i2){
                    if (warehouse_open[i2]){
                        warehouse_open[i1] = true;
                        warehouse_open[i2] = false;
                        double cost = ReassignCustomers();

                        if (cost < current_objective){
                            current_objective = cost;
                            best_open = i1;
                            best_closed = i2;
                        }
```

```
                    warehouse_open[i1] = false;
                    warehouse_open[i2] = true;
                }
            }
        }
    }

    if (best_open != -1 && best_closed != -1){
        warehouse_open[best_open] = true;
        warehouse_open[best_closed] = false;
        return true;
    }

    return false;
}

/**
 * @brief Performs a local search heuristic to optimize the current
solution.
 *
 * @param objective The current objective cost.
 * @return The optimized objective cost after local search.
 *
 * This function applies a local search heuristic, iteratively
attempting to open, close, and
 * open-close warehouses to reduce the objective cost. The search stops
if no improvement is found
 * or the maximum number of local iterations is reached.
 */
double GRASP::LocalSearchHeuristic(double objective){
    current_objective = objective;
    bool improved = true;
    int iterations = 0;

    while (improved){
        improved = false;

        if (CloseWarehouse()){
            improved = true;
            current_objective = ReassignCustomers();
        }

        if (!improved && OpenWarehouse()){
            improved = true;
            current_objective = ReassignCustomers();
        }

        if (!improved && OpenCloseWarehouse()){
```

```
                improved = true;
                current_objective = ReassignCustomers();
            }

            iterations++;
        }

        current_objective = ReassignCustomers();

        return current_objective;
    }

    /**
     * @brief Constructs a greedy randomized solution and applies local
search to it.
     *
     * @return The cost of the constructed solution.
     *
     * This function performs the greedy randomized constructive phase of
the GRASP algorithm.
     * It iteratively opens warehouses based on a restricted candidate list
determined by the parameter alpha,
     * then applies local search to optimize the solution.
     */
    double GRASP::GreedyRandomizedConstructive(){
        double best_cost = MAX_DOUBLE;

        std::fill(warehouse_open.begin(), warehouse_open.end(), false);
        std::fill(best_assignment.begin(), best_assignment.end(), -1);

        while (true){
            int best_warehouse = -1;
            double imp_cost = best_cost;
            std::vector<int> restricted_candidate_list;
            double min_cost = MAX_DOUBLE;
            double max_cost = -MAX_DOUBLE;
            std::vector<double> costs(number_of_warehouses, MAX_DOUBLE);
            std::vector<bool> is_open(number_of_warehouses, false);
            std::vector<std::thread> threads;

            for (int i = 0; i < number_of_warehouses; ++i){
                if (!warehouse_open[i]){
                    threads.push_back(std::thread([&, i]{
                        warehouse_open[i] = true;
                        double cost = ReassignCustomers();
                        warehouse_open[i] = false;
                        costs[i] = cost;
                        is_open[i] = true;
```

```
                }));
            }
        }

        for (auto &th : threads){
            if (th.joinable()){
                th.join();
            }
        }

        for (int i = 0; i < number_of_warehouses; ++i){
            if (is_open[i]){
                if (costs[i] < min_cost)
                    min_cost = costs[i];
                if (costs[i] > max_cost)
                    max_cost = costs[i];
            }
        }

        double threshold = min_cost + alpha * (max_cost - min_cost);

        for (int i = 0; i < number_of_warehouses; ++i){
            if (is_open[i] && costs[i] <= threshold){
                restricted_candidate_list.push_back(i);
            }
        }

        if (restricted_candidate_list.empty())
            break;

        int chosen_index = rand() % restricted_candidate_list.size();
        best_warehouse = restricted_candidate_list[chosen_index];
        warehouse_open[best_warehouse] = true;
        best_cost = ReassignCustomers();

        for (int j = 0; j < number_of_customers; ++j){
            best_assignment[j] = customer_assignment[j];
        }
    }

    return best_cost;
}

/**
 * @brief Solves the given problem using the GRASP algorithm.
 *
 * @param problem The problem instance to solve.
 * @return A vector of pairs representing the assignment of customers
```

```
to warehouses.
     *
     * This function performs the GRASP algorithm over multiple iterations,
each time constructing
     * a greedy randomized solution and applying local search to optimize
it. The best solution found
     * across all iterations is returned as a vector of
customer-to-warehouse assignments.
     */
    std::vector<std::pair<int, int>> GRASP::solve(const Problem &problem){
        initialize(problem);

        double best_cost = MAX_DOUBLE;
        double cost = GreedyRandomizedConstructive();
        cost = LocalSearchHeuristic(cost);

        if (cost < best_cost){
            best_cost = cost;

            for (int j = 0; j < number_of_customers; ++j){
                best_assignment[j] = customer_assignment[j];
            }
        }

        warehouse_open = best_warehouse_open;

        for (int j = 0; j < number_of_customers; ++j){
            customer_assignment[j] = best_assignment[j];
        }

        std::vector<std::pair<int, int>> result;

        for (int i = 0; i < number_of_customers; ++i){
            result.push_back({i, best_assignment[i]});
        }

        return result;
    }
}
```

SimulatedAnnealingAlgorithm.hpp

```
#ifndef SIMULATEDANNEALINGALGORITHM_H
#define SIMULATEDANNEALINGALGORITHM_H
#include <vector>
```

```
#include <random>
#include "../problem/Problem.hpp"
class Problem;
namespace algorithm {

        class SimulatedAnnealingAlgorithm {
        public:
            struct Solution {
                std::vector<int> assignment;
                double total_cost = 0.0;
            };

            SimulatedAnnealingAlgorithm(double initial_temperature, double
final_temperature, double cooling_rate, int iterations_per_temp)
                : initial_temperature(initial_temperature),
final_temperature(final_temperature), cooling_rate(cooling_rate),
iterations_per_temp(iterations_per_temp) {}

            std::vector<std::pair<int, int>> solve(const Problem& problem)
const;

        private:
            double calculateCost(const std::vector<int>& assignment, const
Problem& problem) const;
            Solution generateNeighbor(const Solution& current_solution,
const Problem& problem) const;
            void localSearch(Solution& solution, const Problem& problem,
int tabu_tenure) const;
            Solution adaptivePerturbation(const Solution& current_solution,
const Problem& problem, int iteration) const;

            double initial_temperature;
            double final_temperature;
            double cooling_rate;
            int iterations_per_temp;
        };

        class RandomGenerator {
        public:
            RandomGenerator() : rng(std::random_device{}()), dist(0.0, 1.0)
{}

            double getRandom() {
                return dist(rng);
            }

            int getRandomInt(int max) {
                std::uniform_int_distribution<int> distInt(0, max - 1);
```

```
                return distInt(rng);
            }

        private:
            std::mt19937 rng;
            std::uniform_real_distribution<double> dist;
        };
    }


#endif //SIMULATEDANNEALINGALGORITHM_H
```

SimulatedAnnealingAlgorithm.cpp

```
#include "SimulatedAnnealingAlgorithm.h"
#include <algorithm>
#include <iostream>
#include <unordered_set>

/**
 * Calculates the total cost of the given assignment.
 */
double algorithm::SimulatedAnnealingAlgorithm::calculateCost(const
std::vector<int>& assignment, const Problem& problem) const {
    double total_cost = 0.0;
    const auto& warehouses = problem.getWarehouses();
    const auto& customers = problem.getCustomers();
    std::vector<bool> facilities_open(warehouses.size(), false);

    for (int j = 0; j < assignment.size(); ++j) {
        int warehouse_index = assignment[j];
        total_cost += customers[j].getAllocationCosts()[warehouse_index];
        if (!facilities_open[warehouse_index]) {
            total_cost += warehouses[warehouse_index].getFixedCost();
            facilities_open[warehouse_index] = true;
        }
    }

    return total_cost;
}

/**
 * Generates a neighboring solution by randomly perturbing a small subset
```

```
of the current solution.
 */
algorithm::SimulatedAnnealingAlgorithm::Solution
algorithm::SimulatedAnnealingAlgorithm::generateNeighbor(const Solution&
current_solution, const Problem& problem) const {
    RandomGenerator random;
    Solution new_solution = current_solution;

    int num_perturbations = std::min(problem.getNumberOfCustomers() / 10,
25);
    std::unordered_set<int> perturbed_customers;
    while (perturbed_customers.size() < num_perturbations) {

perturbed_customers.insert(random.getRandomInt(problem.getNumberOfCustomers
()));
    }

    for (int client : perturbed_customers) {
        new_solution.assignment[client] =
random.getRandomInt(problem.getNumberOfWarehouses());
    }

    localSearch(new_solution, problem, 6);
    return new_solution;
}

/**
 * Performs a local search to refine the given solution using a tabu list
to avoid cycles.
 */
void algorithm::SimulatedAnnealingAlgorithm::localSearch(Solution&
solution, const Problem& problem, int tabu_tenure) const {
    const auto& warehouses = problem.getWarehouses();
    const auto& customers = problem.getCustomers();
    std::unordered_set<int> tabu_list;
    int iterations_without_improvement = 0;

    while (iterations_without_improvement < 20) {
        bool found_improvement = false;

        for (int j = 0; j < customers.size(); ++j) {
            int current_warehouse = solution.assignment[j];
            double current_cost =
customers[j].getAllocationCosts()[current_warehouse];
            if (tabu_list.find(current_warehouse) == tabu_list.end()) {
                current_cost +=
warehouses[current_warehouse].getFixedCost();
            }
```

```
            int best_warehouse = current_warehouse;
            double best_cost = current_cost;

            for (int i = 0; i < warehouses.size(); ++i) {
                if (i != current_warehouse && tabu_list.find(i) ==
tabu_list.end()) {
                    double new_cost = customers[j].getAllocationCosts()[i]
+ warehouses[i].getFixedCost();
                    if (new_cost < best_cost) {
                        best_warehouse = i;
                        best_cost = new_cost;
                        found_improvement = true;
                    }
                }
            }

            if (found_improvement) {
                solution.assignment[j] = best_warehouse;
                tabu_list.insert(best_warehouse);
                if (tabu_list.size() > tabu_tenure) {
                    tabu_list.erase(tabu_list.begin());
                }
                iterations_without_improvement = 0;
            } else {
                iterations_without_improvement++;
            }
        }

        if (!found_improvement) {
            iterations_without_improvement++;
        }
    }

    solution.total_cost = calculateCost(solution.assignment, problem);
}

/**
 * Generates a new solution by adaptively perturbing a portion of the
current solution based on the iteration number.
 */
algorithm::SimulatedAnnealingAlgorithm::Solution
algorithm::SimulatedAnnealingAlgorithm::adaptivePerturbation(const
Solution& current_solution, const Problem& problem, int iteration) const {
    RandomGenerator random;
    Solution new_solution = current_solution;

    int num_customers = problem.getNumberOfCustomers();
```

```
    int perturbation_size = (iteration % 2 == 0) ? num_customers / 3 :
num_customers / 5;

    std::unordered_set<int> perturbed_customers;
    while (perturbed_customers.size() < perturbation_size) {
        perturbed_customers.insert(random.getRandomInt(num_customers));
    }

    for (int client : perturbed_customers) {
        new_solution.assignment[client] =
random.getRandomInt(problem.getNumberOfWarehouses());
    }

    return new_solution;
}

/**
 * Solves the problem using the simulated annealing algorithm.
 */
std::vector<std::pair<int, int>>
algorithm::SimulatedAnnealingAlgorithm::solve(const Problem& problem) const
{
    RandomGenerator random;
    int num_warehouses = problem.getNumberOfWarehouses();
    int num_customers = problem.getNumberOfCustomers();

    Solution initial_solution;
    initial_solution.assignment.resize(num_customers);
    for (int j = 0; j < num_customers; ++j) {
        initial_solution.assignment[j] =
random.getRandomInt(num_warehouses);
    }
    initial_solution.total_cost =
calculateCost(initial_solution.assignment, problem);

    Solution current_solution = initial_solution;
    Solution best_solution = current_solution;

    double temperature = initial_temperature;
    int iteration = 0;

    while (temperature > final_temperature) {
        for (int i = 0; i < iterations_per_temp; ++i) {
            Solution new_solution = generateNeighbor(current_solution,
problem);
            new_solution.total_cost =
calculateCost(new_solution.assignment, problem);
            localSearch(new_solution, problem, 6);
```

```
            double delta_cost = new_solution.total_cost -
current_solution.total_cost;

            if (delta_cost < 0 || std::exp(-delta_cost / temperature) >
random.getRandom()) {
                current_solution = new_solution;
            }

            if (current_solution.total_cost < best_solution.total_cost) {
                best_solution = current_solution;
            }
        }

        temperature *= cooling_rate;

        if (++iteration % 30 == 0) {
            current_solution = adaptivePerturbation(best_solution, problem,
iteration);
            current_solution.total_cost =
calculateCost(current_solution.assignment, problem);
        }
    }

    std::vector<std::pair<int, int>> result(num_customers);
    for (int j = 0; j < num_customers; ++j) {
        result[j] = std::make_pair(j, best_solution.assignment[j]);
    }

    return result;
}
```

HillClimbing.hpp

```
#pragma once
#include "Algorithm.hpp"
#include <vector>

namespace algorithm {

class HillClimbingAlgorithm : public Algorithm {
public:
    std::vector<std::pair<int, int>> solve(const Problem& problem) const
override;
```

```
private:
    double calculateCost(const Problem& problem, const std::vector<bool>&
openWarehouses) const;
    void getInitialSolution(const Problem& problem, std::vector<bool>&
openWarehouses) const;
    void getBestNeighbor(const Problem& problem, const std::vector<bool>&
currentSolution, std::vector<bool>& bestNeighbor) const;
};

}
```

HillClimbing.cpp

```
#include "HillClimbingAlgorithm.hpp"
#include <limits>
#include <algorithm>
#include <chrono>
#include <iostream>
#include <iomanip>

namespace algorithm {

double HillClimbingAlgorithm::calculateCost(const Problem& problem, const
std::vector<bool>& openWarehouses) const {
    double totalCost = 0.0;

    const auto& warehouses = problem.getWarehouses();
    const auto& customers = problem.getCustomers();

    for (size_t i = 0; i < openWarehouses.size(); ++i) {
        if (openWarehouses[i]) {
            totalCost += warehouses[i].getFixedCost();
        }
    }

    for (const auto& customer : customers) {
        double minCost = std::numeric_limits<double>::max();
        const auto& allocationCosts = customer.getAllocationCosts();

        for (size_t i = 0; i < openWarehouses.size(); ++i) {
            if (openWarehouses[i]) {
                minCost = std::min(minCost, allocationCosts[i]);
            }
```

```
        }

        totalCost += minCost;
    }

    return totalCost;
}

void HillClimbingAlgorithm::getInitialSolution(const Problem& problem,
std::vector<bool>& openWarehouses) const {
    // Initially open all warehouses
    openWarehouses.assign(problem.getNumberOfWarehouses(), true);
}

void HillClimbingAlgorithm::getBestNeighbor(const Problem& problem, const
std::vector<bool>& currentSolution, std::vector<bool>& bestNeighbor) const
{
    double bestCost = std::numeric_limits<double>::max();
    bestNeighbor = currentSolution;

    for (size_t i = 0; i < currentSolution.size(); ++i) {
        std::vector<bool> neighbor = currentSolution;
        neighbor[i] = !neighbor[i]; // Toggle the state of the ith
warehouse

        double neighborCost = calculateCost(problem, neighbor);
        if (neighborCost < bestCost) {
            bestCost = neighborCost;
            bestNeighbor = neighbor;
        }
    }
}

std::vector<std::pair<int, int>> HillClimbingAlgorithm::solve(const
Problem& problem) const {
    std::vector<bool> currentSolution;
    getInitialSolution(problem, currentSolution);

    double currentCost = calculateCost(problem, currentSolution);
    bool localOptimum = false;

    int iteration = 0;

    while (!localOptimum) {
        ++iteration;

        auto start = std::chrono::high_resolution_clock::now();
        std::vector<bool> neighborSolution;
```

```
            getBestNeighbor(problem, currentSolution, neighborSolution);

            double neighborCost = calculateCost(problem, neighborSolution);

            if (neighborCost < currentCost) {
                currentSolution = neighborSolution;
                currentCost = neighborCost;
            } else {
                localOptimum = true;
            }

            auto end = std::chrono::high_resolution_clock::now();
            std::chrono::duration<double> elapsed = end - start;

            std::cout << "Iteration " << iteration << " time: " << std::fixed
<< std::setprecision(2) << elapsed.count() << "s. New total cost: " <<
currentCost << std::endl;
        }

        std::vector<std::pair<int, int>> assignments;
        const auto& customers = problem.getCustomers();

        for (size_t i = 0; i < customers.size(); ++i) {
            const auto& allocationCosts = customers[i].getAllocationCosts();
            int bestWarehouse = -1;
            double minCost = std::numeric_limits<double>::max();

            for (size_t j = 0; j < currentSolution.size(); ++j) {
                if (currentSolution[j] && allocationCosts[j] < minCost) {
                    minCost = allocationCosts[j];
                    bestWarehouse = static_cast<int>(j);
                }
            }

            assignments.emplace_back(static_cast<int>(i), bestWarehouse);
        }

        return assignments;
    }

    } // namespace algorithm
```