Busca Local Iterada aplicada ao Problema do Ensopado Perfeito

INF05010 - Otimização Combinatória

Mateus Nunes Campos

Agosto 2024

Abstract

Este trabalho tem como objetivo implementar a meta-heurística de Busca Local Iterada para resolver o Problema do Ensopado Perfeito. O problema é uma variação do problema da mochila, com restrições adicionais de compatibilidade entre os itens. A meta é encontrar uma combinação de ingredientes que maximize o sabor total, respeitando o peso máximo permitido e evitando combinações de ingredientes incompatíveis.

1 Introdução

O problema do Ensopado Perfeito é uma variação do problema da mochila, onde temos restrições adicionais de incompatibilidade entre os itens. O objetivo é encontrar uma combinação de ingredientes que maximize o sabor total, respeitando um limite de peso e evitando combinações incompatíveis.

Formalmente, o problema é definido da seguinte maneira:

1.1 Instância

Dado um conjunto de n ingredientes, onde cada ingrediente i possui um peso $w_i \geq 0$ e um sabor $t_i \geq 0$, e uma lista de incompatibilidades I, onde cada $i \in I$ é um par $i = (j, k)$ de ingredientes que não podem ser usados juntos.

1.2 Solução

Uma seleção de ingredientes de modo que nenhum par de ingredientes incompatíveis seja selecionado e que o peso total dos ingredientes selecionados não ultrapasse W.

1.3 Objetivo

Maximizar o sabor total, ou seja, a soma dos sabores dos ingredientes selecionados.

2 Formulação do Problema

$$\text{Maximize} \quad \sum_{i=1}^{n} t_i x_i \tag{1}$$

$$\text{Sujeito a:} \quad \sum_{i=1}^{n} w_i x_i \leq W \tag{2}$$

$$x_j + x_k \leq 1 \quad \forall (j,k) \in I \tag{3}$$

$$x_i \in \{0,1\} \quad \forall i \in \{1,\ldots,n\} \tag{4}$$

onde:

- x_i é uma variável binária que indica se o ingrediente i foi selecionado (1) ou não (0).
- w_i e t_i são o peso e o sabor do ingrediente i, respectivamente.
- W é o peso máximo permitido para o ensopado.
- I é o conjunto de pares de ingredientes incompatíveis.

Explicação das Restrições:

1. **Restrição 1:** $\sum_{i=1}^{n} w_i x_i \leq W$
    - Garante que o peso total dos ingredientes selecionados não exceda o peso máximo permitido W.
2. **Restrição 2:** $x_j + x_k \leq 1 \quad \forall (j,k) \in I$
    - Impede que dois ingredientes incompatíveis (representados pelo par $(j,k) \in I$) sejam selecionados simultaneamente.

- A restrição (1) não se expande para mais restrições; ela escala apenas com o número total de ingredientes n.
- A restrição (2) se expande para $|I|$ restrições, onde $|I|$ é o número de pares de ingredientes incompatíveis.
- A restrição (3) se expande para n restrições.

Desta forma, temos um total de n variáveis e $|I|+n$ restrições. Esta formulação revela a complexidade do problema à medida que o número de ingredientes n e o número de pares incompatíveis $|I|$ aumentam.

3 Busca Local Iterada

A Busca Local Iterada (ILS) é uma meta-heurística eficaz para resolver problemas de otimização combinatória complexos. Sua principal característica é a combinação de busca local intensiva com perturbações , permitindo que o algoritmo escape de ótimos locais e explore melhor o espaço de soluções. Neste trabalho, diversas técnicas foram implementadas para melhorar o desempenho e o funcionamento da meta-heurística, tendo sido utilizada a linguagem C++.

3.1 Representação de uma Solução

Uma solução é representada como um vetor de ingredientes selecionados, onde cada elemento desta lista corresponde a um ingrediente específico. A seleção deve respeitar as restrições de incompatibilidade entre pares de ingredientes e a soma dos pesos dos ingredientes não pode exceder o limite W.

3.2 Geração da Solução Inicial

A geração da solução inicial é baseada em uma heurística de pontuação para cada ingrediente. Essa pontuação é calculada como o sabor do ingrediente dividido pelo seu peso. Ingredientes com uma alta relação sabor/peso são preferencialmente incluídos na solução inicial.

O processo de construção da solução inicial consiste em iterar sobre a lista de ingredientes ordenada pela pontuação decrescente e incluir os ingredientes na solução, desde que sejam compatíveis com os já selecionados e que o limite de peso W não seja ultrapassado. Este método permite que a solução inicial tenha uma qualidade superior em relação a uma escolha completamente aleatória, uma vez que tende a selecionar ingredientes que maximizam o sabor com o menor peso.

3.3 Geração de Vizinhanças

A vizinhança de uma solução é gerada realizando pequenas alterações na solução corrente. Existem duas principais operações para gerar vizinhanças:

1. **Substituição**: Trocar um ingrediente da solução por outro que seja compatível e que respeite a restrição de peso. Esta troca não considera diretamente o sabor do novo ingrediente, focando apenas na compatibilidade e no peso.

2. **Adição**: Caso o peso atual permita, um novo ingrediente pode ser adicionado à solução, sem a necessidade de remover outro ingrediente. Esta operação é realizada até não haver mais ingredientes compatíveis, ou não haver mais peso livre.

Essas operações permitem explorar diferentes soluções próximas, mantendo o foco em regiões promissoras do espaço de busca.

3.4 Critério de Parada

O algoritmo é executado por um número predefinido de iterações X. Após X iterações, o processo de busca é interrompido, e a melhor solução encontrada até aquele momento é retornada como a solução final.

3.5 Perturbações e Critério de Aceitação

Perturbações são aplicadas à solução corrente para escapar de ótimos locais. A estratégia de perturbação escolhida consiste em remover aleatoriamente um par de ingredientes da solução. Essa remoção cria espaço para a inclusão de novos ingredientes, potencialmente gerando combinações ainda não exploradas.

O algoritmo também mantém um contador de iterações sem melhoria. Se este contador ultrapassar o parâmetro de threshold, perturbações mais fortes são realizadas, removendo dois pares de ingredientes da solução. Existem dois breakpoints no algoritmo: 50 e 99 iterações sem melhoria. No caso de 50 iterações sem melhoria, são removidos três pares de ingredientes; já no caso de 99 iterações, são removidos quatro pares de ingredientes. Esse contador é reiniciado ao chegar em 100 iterações, independentemente de ter havido melhoria ou não.

O objetivo de mudar a força da perturbação dinamicamente é que, dependendo da intensidade da perturbação, ela pode tanto resetar a solução quanto, se for muito fraca, não ser capaz de sair do ótimo local. Portanto, são realizadas diversas perturbações fracas, médias e as ocasionais fortes.

4 Algoritmo

Algorithm 1 Iterated Local Search

1: $s_0 \leftarrow$ *Generate Initial Solution*
2: $s^* \leftarrow$ *localSearch*(s_0)
3: **repeat**
4: $s' \leftarrow$ *perturbate*(s^*, *no_improvement_count*)
5: $s'^* \leftarrow$ *localSearch*(s')
6: $s^* \leftarrow$ *acceptanceCriterion*(s^*, s'^*)
7: iterations++
8: **until** iterations = *MAX_ITERATIONS*

4.1 Parâmetros

- **neighborhood_size**: Número de vizinhos gerados dentro da função *localSearch*.
- **max_iterations**: Número máximo de iterações que o algoritmo irá executar. O algoritmo para quando atinge esse número.
- **no_improvement_threshold**: Número de iterações sem melhoria na função objetivo necessárias para aplicar perturbações mais fortes.
- **max_iterations_without_improvements**: Este parâmetro é utilizado apenas se a instância do problema for muito lenta. O algoritmo irá parar se atingir esse número de iterações sem melhoria. Geralmente, este parâmetro está desativado.

Os parâmetros serão mais detalhados nas próximas seções.

5 Implementação

5.1 Plataforma de Implementação

O trabalho foi implementado e testado em um sistema operacional Windows 11 Pro (64-bit), com um processador Intel(R) Core(TM) i5-10400F CPU, com clock base de 2.9 GHz e boost de até 4.3 GHz, 6 núcleos e 12 threads. O processador possui cache L1 de 384 KB, L2 de 1.5 MB e L3 de 12 MB, além de 16 GB de memória RAM. A linguagem de programação utilizada foi C++, com o compilador MinGW.

5.2 Estruturas de Dados Utilizadas

Como os ingredientes são representados por números inteiros, temos uma variável global que armazena a quantidade de ingredientes da instância. Seus sabores e pesos são armazenados em dois vetores de tamanho igual ao de ingredientes, o que torna o acesso constante, pois basta acessar o índice respectivo ao ingrediente.

Os pares de ingredientes incompatíveis são armazenados em um `unordered_set` da biblioteca padrão do C++, que armazena um `pair<int, int>` (também da biblioteca padrão), com a função de hash respectiva ao par. Isso torna a checagem de incompatibilidade extremamente eficiente, pois evita a necessidade de iterar por um vetor inteiro testando.

6 Geração de Vizinhanças

A operação mais custosa do algoritmo é, de longe, a geração de vizinhanças. Ao remover um ingrediente, o algoritmo precisa encontrar outro ingrediente viável para substituí-lo.

Além disso, ao realizar as perturbações, ingredientes são removidos da solução, o que implica em um maior trabalho para a geração de vizinhos, pois é necessário adicionar novos ingredientes até não haver mais ingredientes válidos.

Isso implica que, ao introduzir um novo ingrediente e o peso não alcançar o limite, é necessário iterar por todos os ingredientes para esgotar possíveis soluções. Para combater isso, após a substituição inicial de ingredientes, um vetor de ingredientes possíveis de serem introduzidos é calculado. Se este vetor não for vazio, um ingrediente aleatório é introduzido, e o vetor é recalculado apenas pelo novo ingrediente introduzido, removendo algum ingrediente incompatível ou que excederia o peso, caso fosse introduzido. Isso evita iterar por todos os ingredientes novamente.

7 Paralelização

É fácil perceber que a geração de vizinhos pode ser paralelizada, visto que não há dependências entre um vizinho e outro. A geração de vizinhos foi paralelizada criando chunks de tamanho baseado no número de threads da máquina. No final, as threads se juntam para armazenar cada vizinho no vetor de vizinhança. Essa paralelização foi realizada usando a biblioteca `future` da biblioteca padrão.

8 Testes de Parâmetros

Os seguintes testes foram realizados utilizando a instância `ep01.dat`, fornecida pelo professor. Cada teste foi executado 10 vezes, com sementes aleatórias diferentes. Foram medidos o tempo de execução e a melhor solução encontrada.

8.1 Teste do parâmetro `neighborhood_size`

Para realizar este teste, fixamos `max_iterations` em 500 e `no_improvement_threshold` em 15. O tamanho da vizinhança foi iniciado em 5 e foi incrementado para 15, 25, 50 e 100, cada vez sendo realizado 10 testes, e levando em consideração o melhor tempo e solução (devido ao fator aleatório). O objetivo é descobrir o parâmetro ideal para o tamanho da vizinhança, considerando o custo computacional para gerá-la.

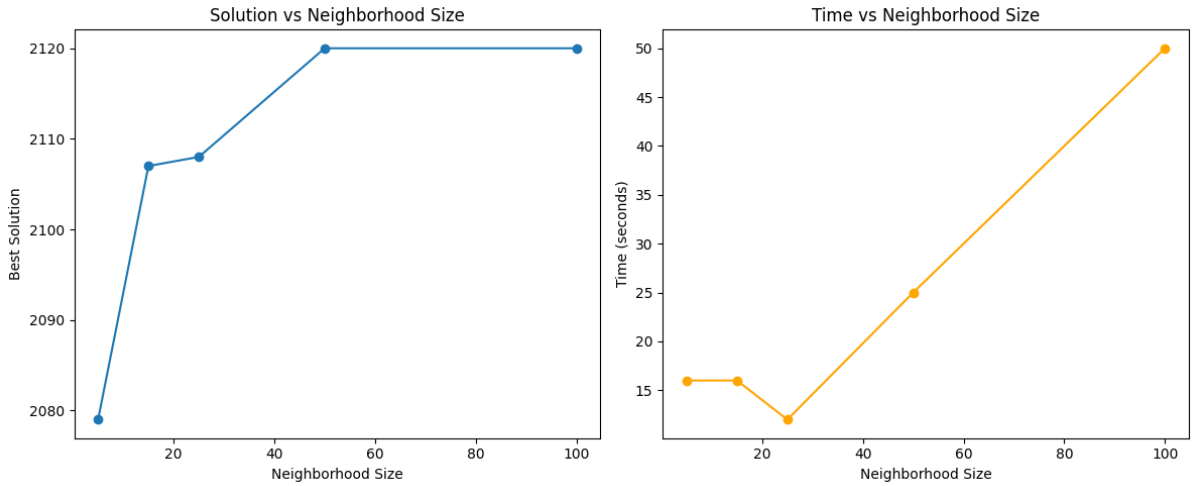

Figure 1: Resultados do teste de `neighborhood_size`.

É fácil perceber que vizinhanças muito pequenas não permitem ao algoritmo explorar soluções diferentes o suficiente, e o overhead de paralelização acaba influenciando no tempo. Já vizinhanças muito grandes impactam no tempo de computação e possuem *diminishing returns* em relação à qualidade das soluções, pois começam a ter vizinhos repetidos na vizinhança. Portanto, o ideal é utilizar um valor médio neste parâmetro, como 25 ou 50, que oferece um bom balanço entre tempo e qualidade da solução.

8.2 Teste do parâmetro `max_iterations`

Para este teste, o parâmetro `neighborhood_size` foi fixado em 25 e `no_improvement_threshold` foi fixado em 15. O número máximo de iterações (`max_iterations`) foi iniciado em 50, e incrementado para 500, 1500, 3000, 5000 e 10000, cada vez sendo realizado 10 testes, levando em consideração o melhor tempo e solução (devido ao fator aleatório). O objetivo é medir o impacto que grandes iterações podem causar na solução final e no tempo de execução.

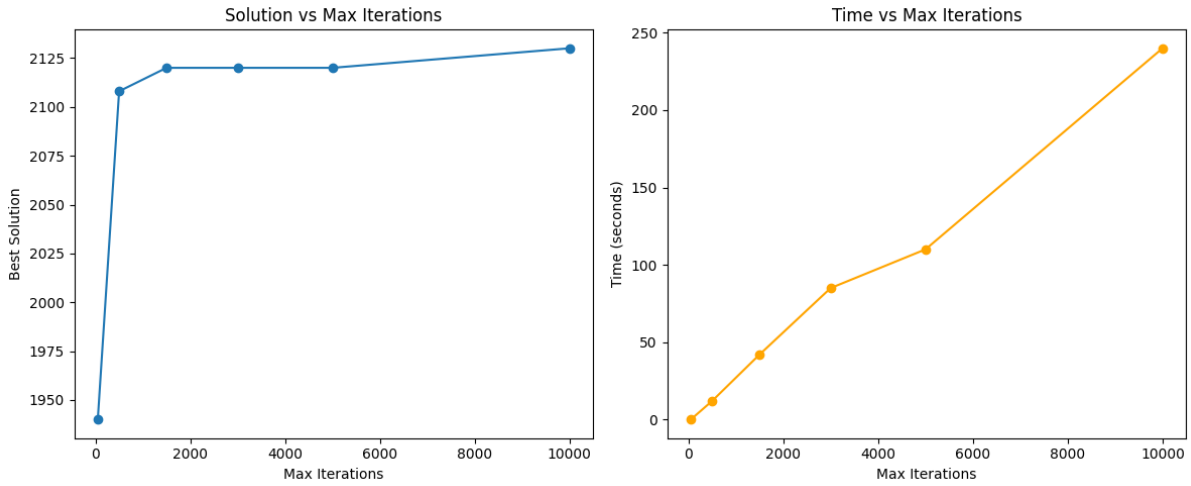

Figure 2: Resultados do teste de `max_iterations`.

Como podemos observar no gráfico, o tempo cresce de forma linear com o número de iterações. Já a qualidade das soluções tem um *diminishing returns* muito alto, mas para acessar boas soluções, o algoritmo precisa rodar, de fato, muitas iterações, como pode ser observado, com 10.000 iterações uma solução melhor foi encontrada.

8.3 Teste do parâmetro `no_improvement_threshold`

Neste teste, `neighborhood_size` foi fixado em 25 e `max_iterations` foi fixado em 5000. O parâmetro `no_improvement_threshold` foi iniciado em 3, e incrementado para 5, 10, 15, 25 e 50. O objetivo é avaliar se é possível melhorar a solução aplicando perturbações fortes mais frequentemente, ou se isso piora a solução. Neste teste, o tempo de execução não sofreu variação significativa.

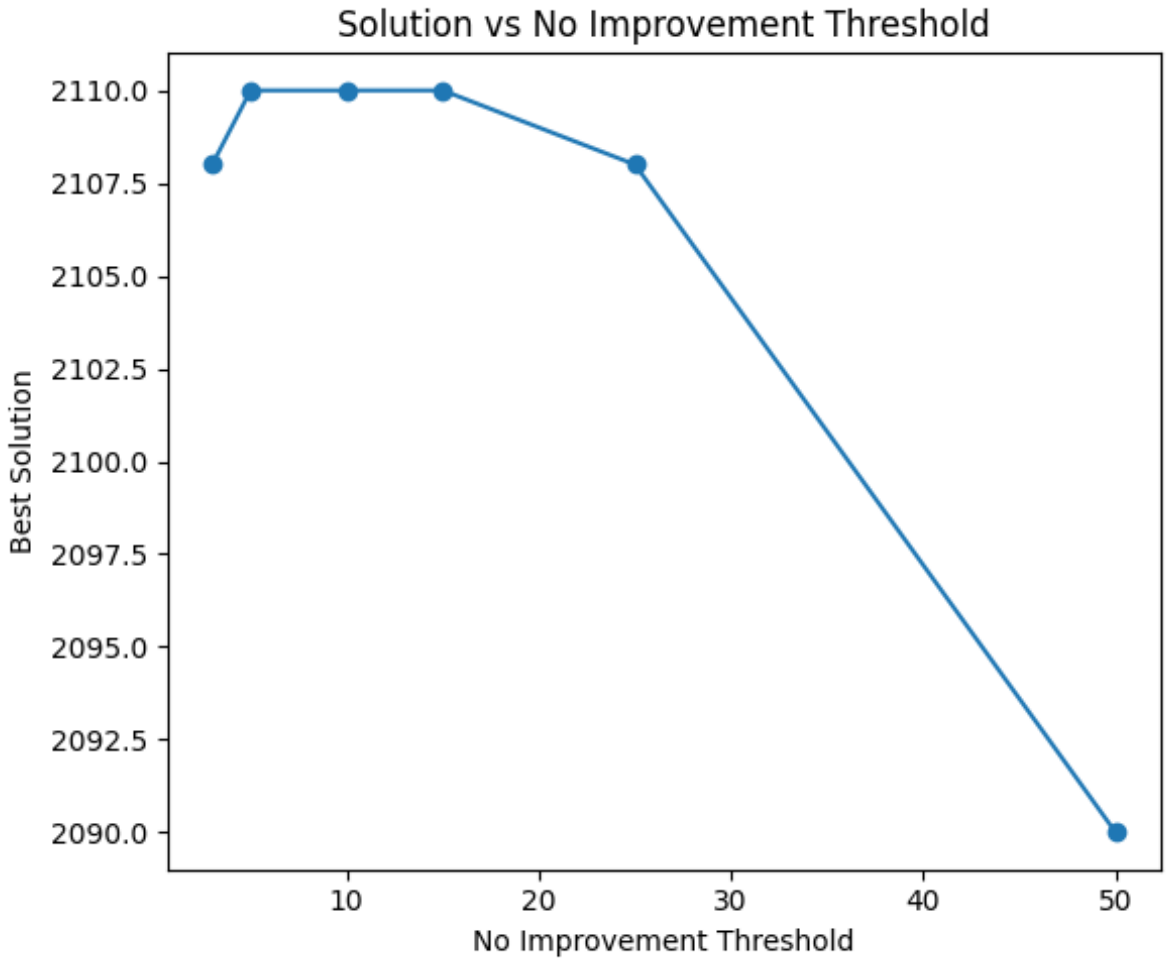

Figure 3: Resultados do teste de `no_improvement_threshold`.

Podemos observar no gráfico que realizar perturbações fortes com muita frequência resulta em uma queda na qualidade da solução, mas se demorar demais, a queda é extremamente alta. Portanto, podemos concluir que o ideal é um número médio de perturbações fortes.

9 Testes das Instâncias

Foram realizados testes para todas as instâncias fornecidas pelo professor, ep01.dat até a instância ep10.dat. Os testes foram realizados com os parâmetros `neighborhood_size` = 25, `max_iterations` = 10.000 e `no_improvement_threshold` = 15, e cada teste foi rodado 5 vezes. Como observado nos testes fixos de cada parâmetro, esses valores apresentaram o melhor desempenho em relação ao balanço entre qualidade de solução e tempo de computação. A partir da instância **ep06.dat** foi necessário utilizar o parâmetro **max_iterations_without_improvements** = 1000, devido a inviabilidade de tempo

para executar tantas iterações em instâncias largas, pois o tempo de processamento da vizinhança era muito extenso. Os testes com a formulação do programa linear foram realizados utilizando o solver Gurobi e rodaram por até 30 minutos.

A tabela abaixo contém as informações sobre as instâncias e BKV (best known value) fornecido pelo professor, onde n é o número de ingredientes, $|I|$ é a quantidade de pares incompatíveis e W é o peso máximo permitido no ensopado.

Instância	n	$\|I\|$	W	BKV
ep01	500	25,144	1,800	2,118
ep02	500	50,012	1,800	1,378
ep03	1,000	29,700	2,000	2,850
ep04	1,000	39,900	2,000	2,730
ep05	1,000	49,855	2,000	2,624
ep06	1,500	89,940	4,000	4,690
ep07	1,500	179,880	4,000	4,440
ep08	2,000	79,960	4,000	5,020
ep09	2,000	239,880	4,000	4,568
ep10	2,000	399,800	4,000	4,390

Table 1: Informações sobre as instâncias.

9.1 Testes com o Gurobi

Instância	**Status**	**Valor da Solução (Sabor Total)**	**BKV**
ep01.dat	TIME_LIMIT	2090.0	2118
ep02.dat	TIME_LIMIT	1378.0	1378
ep03.dat	TIME_LIMIT	2820.0	2850
ep04.dat	TIME_LIMIT	2710.0	2730
ep05.dat	TIME_LIMIT	2580.0	2624
ep06.dat	TIME_LIMIT	4650.0	4690
ep07.dat	TIME_LIMIT	4410.0	4440
ep08.dat	TIME_LIMIT	4970.0	5020
ep09.dat	TIME_LIMIT	4520.0	4568
ep10.dat	TIME_LIMIT	4330.0	4390

Table 2: Resultados das Instâncias com o solver

Podemos observar que em nenhuma instância o solver conseguiu atingir a solução ótima dentro dos 30 minutos disponíveis. Além disso, somente na instância `ep02.dat` o solver foi capaz de alcançar o BKV.

9.2 Testes com o Iterated Local Search

Instância	SI	MSF	BSF	BKV	% Desvio (SI-BSF)	% Desvio (BKV)	Tempo Médio
ep01	669	2124	2130	2118	-218.21	0.57	5min 40s
ep02	267	1338.8	1391	1378	-409.73	0.93	5min 42s
ep03	1944	2854	2860	2850	-47.06	0.35	21min 55s
ep04	1505	2722	2730	2730	-81.33	0.00	20min 24s
ep05	1342	2603.3	2629	2624	-95.97	0.19	18min 24s
ep06*	4534	4610.2	4640	4690	-2.34	-1.07	9min 21s
ep07*	2334	4398	4410	4440	-88.94	-0.68	12min 52s
ep08*	4916	4970	5010	5020	-1.91	-0.20	39min 31s
ep09*	3634	4513	4548	4568	-25.16	-0.44	29min 18s
ep10*	1855	4345.8	4360	4390	-135.09	-0.68	16min 28s

Table 3: Resultados dos testes do Iterated Local Search. SI: valor da solução inicial, MSF: solução final média, BSF: melhor solução final, BKV: melhor valor conhecido.

9.3 Análise

A Tabela 3 resume os resultados dos testes realizados utilizando a implementação do Iterated Local Search em diferentes instâncias do problema. Para cada instância, foram registrados os valores da solução inicial (SI), da solução final (SF), do melhor valor conhecido (BKV), o desvio percentual entre SI e SF, o desvio percentual entre SF e BKV, e o tempo de execução até a parada. As instâncias marcadas com * utilizaram o parâmetro de parada sem melhoria.

9.4 Desempenho do Algoritmo (SI vs SF)

Em todas as instâncias, a solução final (SF) encontrada pelo ILS superou significativamente a solução inicial (SI), conforme indicado pelos desvios percentuais negativos entre SI e SF. Isso demonstra que o ILS foi eficaz em melhorar a qualidade da solução ao longo das iterações.

Por exemplo, na instância ep01, a SF é 2130, um aumento de 218,21% em relação ao SI de 669, mostrando que o algoritmo conseguiu explorar o espaço de busca para encontrar soluções de melhor qualidade.

9.5 Comparação com o Melhor Valor Conhecido (SF vs BKV)

Uma parte das soluções finais (SF) são melhores que os BKVs fornecidos, em especial as das instâncias mais simples. A degradação de soluções começou após a instância ep06, pois

o tempo de computação da vizinhança começou a ficar custoso devido ao tamanho do vetor de incompatibilidades. Logo, as soluções finais destas instâncias grandes foram inferiores aos BKVs fornecidos. No geral, se formos comparar com as soluções apresentadas pelo solver, a implementação da meta-heurística encontrou soluções melhores em menos tempo.

9.6 Tempo de Execução e Estado de Finalização

O tempo de execução variou significativamente entre as instâncias, refletindo a complexidade e o tamanho de cada problema. Instâncias como ep08 e ep09 demandaram mais tempo (40min 27s e 28min 3s, respectivamente), o que é esperado para problemas maiores ou mais complexos. Vale lembrar que essas instâncias utilizaram o parâmetro de parada após iterações sem melhoria, e ainda assim alcançaram tempos significativos.

9.7 Conclusão

O Iterated Local Search se mostrou eficaz na melhoria das soluções iniciais e em sua aproximação aos melhores valores conhecidos. No entanto, em alguns casos, o tempo limitado ou os limites de iterações podem ter impedido o algoritmo de encontrar a solução ótima global.

Ajustes nos parâmetros, como o número de iterações ou a estratégia de perturbação, poderiam ser explorados para melhorar ainda mais os resultados, especialmente em instâncias onde o desvio percentual em relação ao BKV foi mais elevado. Uma implementação mais refinada da geração de vizinhança poderia surtir grandes efeitos na velocidade do programa, pois este foi o maior gargalo encontrado.

No geral, a implementação foi bem-sucedida, mas há espaço para otimizações no código.

References

[1] Lourenço, H.R., Martin, O., and Stützle, T. (2001). A beginner's introduction to Iterated Local Search. In *Proceedings of the 4th Metaheuristics International Conference*, Porto, Portugal, pp. 1-11.

[2] Lourenço, H.R., Martin, O., and Stützle, T. (2010). Iterated Local Search: Framework and Applications. In *Handbook of Metaheuristics, 2nd Edition*, Vol. 146, M. Gendreau and J.Y. Potvin (eds.), Springer New York, International Series in Operations Research & Management Science, pp. 363-397. ISBN: 978-1-4419-1663-1.

[3] Pferschy, U., and Schauer, J. (2017). Approximation of knapsack problems with conflict and forcing graphs. *Journal of Combinatorial Optimization*, 33, 1300–1323. https://doi.org/10.1007/s10878-016-0035-7