Algoritmo Genético para o Problema do Ensopado Perfeito

Gustavo Prolla Lacroix
gustavo.lacroix@ufrgs.br

Tomás Mitsuo Ueda
tomas.ueda@ufrgs.br

Agosto 2024

1 Introdução

Este trabalho tem como objetivo implementar uma meta-heurística que resolve o problema do Ensopado Perfeito. A abordagem utilizada é Algoritmos Genéticos. O problema pode ser definido da seguinte maneira:

Instância: Temos n ingredientes, e cada ingrediente i tem um peso $w_i \geq 0$ e um sabor $t_i \geq 0$. Há ainda uma lista de ingredientes incompatíveis I, em que cada $i \in I$ é um par $i = (j, k)$ de ingredientes que não podem ser usados juntos.

Solução: Uma seleção de ingredientes, de modo que nenhum par de ingredientes incompatíveis tenha sido selecionado e que o peso total dos ingredientes selecionados não ultrapasse W.

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

2 Formulação Programa Inteiro

Variáveis: $x_i \in \{0, 1\}$, tal que $x_i = 1$ se o alimento (i) está no ensopado, e $x_{ij} = 0$ se ela não está.
Função Objetivo:

$$\max \sum_{i \in [n]} t_i x_i$$

Restrições:

$$\sum_{i \in [n]} w_i x_i <= W \tag{1}$$

$$x_j + x_k <= 1 \quad \forall (j, k) \in I \tag{2}$$

3 Parâmetros

population_size: tamanho da população inicial, $p \in [1, n!]$
recombination_rate: taxa de recombinação, $r \in [0, 1]$
beta_rank: pressão seletiva na recombinação, $\beta_{rank} \in [1, 2]$
mutation_rate: taxa de mutação, $m \in [0, 1]$
max_generations: número máximo de gerações, $g \in [0, \text{inf}]$
max_non_improving_generations: número máximo de gerações não melhorantes $ng \in [0, \text{inf}]$
seq_non_improving_generations: booleano, se verdadeiro são necessárias ng gerações não melhorantes consecutivas para terminar
max_time: tempo máximo de execução
seed: semente para o PRNG

4 Algoritmo

Algorithm 1 Algoritmo Genético

```
populacao ← geraPopulaçãoInicial()
while !criterioDeTerminaçãoAlcançado() do
    nova_populacao ← ∅
    while len(nova_populacao) < p do
        p1, p2 ← seleção_recombinação(populacao)
        if random() < r then
            f ← recombinação(p1, p2)
        else
            f ← max(p1, p2)
        end if
        if random() < m then
            mutate(f)
        end if
        nova_populacao ← nova_populacao ∪ {f}
    end while
    populacao ← nova_populacao
end while
```

4.1 Geração da População Inicial

Geramos p soluções aleatórias distintas. Escolhemos ingredientes válidos, isto é, que são compatíveis e que cabem no ensopado, até que não haja mais ingredientes válidos.

4.2 Escolha dos indivíduos para recombinação

Realizamos um linear Rank-based Selection [1]. O parâmetro β_{rank} define a pressão seletiva, isto é, o número esperado de filhos do melhor indivíduo a cada geração. Rank-based Selection evita uma convergência prematura causada por indivíduos muito acima da média populacional, ou seja, permite que indivíduos piores reproduzam e portanto melhorem.

4.3 Recombinação

Realizamos um Uniform Crossover factibilizado. Cada ingrediente tem 50% de chance de vir de um dos pais. Apenas adicionamos um ingrediente se ele satisfaz as restrições do problema.

4.4 Mutação

A nossa mutação consiste de uma busca local randomizada, seguida de uma forma de k-EM. Primeiro escolhemos ingredientes válidos para adicionar aleatoriamente, como fazemos na geração da população inicial. Seja a o número de ingredientes adicionados. Por fim, retiramos aleatoriamente um ingrediente e adicionamos aleatoriamente um ingrediente válido $0.1n + (max(0, 0.1n - a))$ vezes. Ao longo deste processo, verificamos se alguma das soluções intermediárias é a melhor solução vista até o momento. Caso seja, atualizamos a melhor solução.

Dizemos que é uma forma de k-EM, pois podemos pensar que k é um número entre $0.1n$ e $0.2n$ e a troca consiste em trocar um ingrediente que está na sopa por outro que não está.

4.5 Terminação

O algoritmo termina nos seguintes casos: tempo máximo atingido, número máximo de gerações atingido, número máximo de gerações não melhorantes atingido, número máximo de gerações não melhorantes consecutivas atingido.

5 Implementação

Implementamos o algoritmo utilizando a linguagem Python 3.10.12 junto com a biblioteca open source *bitarray* 2.9.2 [2]

5.1 Estrutura de dados

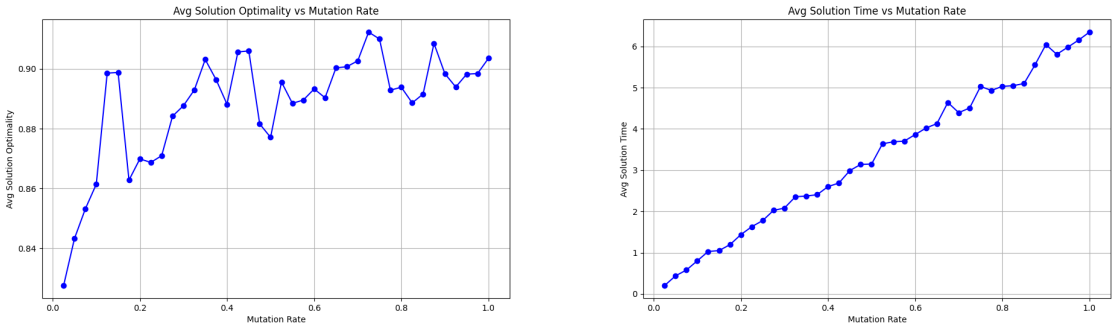
Utilizamos um bitarray para guardar os bits de uma solução, um mapa da solução em forma de matriz/grafo (utilizado para testar se uma solução é válida), e a matriz de incompatibilidades. A operação de mutação é realizada in-place, sendo criada uma cópia caso seja a melhor solução.

6 Testes de parâmetros

Esse trabalho foi testado em sistema operacional Ubuntu 22.04, com um processador AMD Ryzen 7 5800G de 8 núcleos e 16GB de memória DDR4. Os testes foram executados em paralelo, com um teste por núcleo. Para realizar os testes utilizamos as instâncias *ep01.dat* e *ep02.dat* visto que são as menores instâncias disponibilizadas. Cada teste foi realizado com 5 sementes diferentes ($s \in [5]$). Normalizamos o eixo-y para ser a média entre todas sementes e instâncias do parâmetro analisado.

6.1 *mutation_rate*

Primeiro testamos a taxa de mutação. Utilizamos um *population_size* de 50 e *max_non_improving_generations* de 10 para que os testes rodassem rápido. Setamos $\beta_{rank} = 2.0$ e $r = 0.0$. Variamos m de 0.025 até 1.0 com incrementos de 0.025



(a) Otimalidade

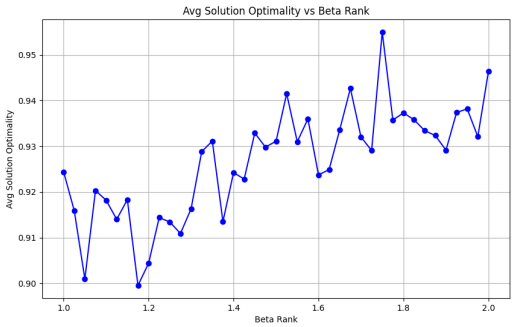
(b) Tempo

Figure 1: Taxa de mutação

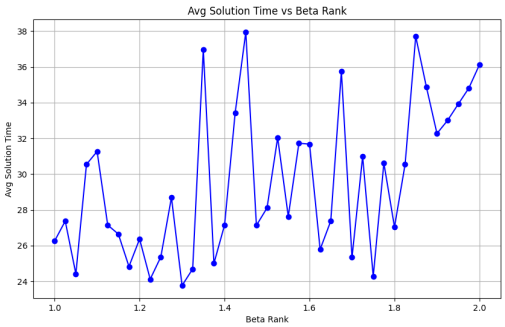
Podemos observar um crescimento logarítmico da otimalidade, comparado com um crescimento linear do tempo.

6.2 *beta_rank*

Utilizamos *population_size* de 50, *max_non_improving_generations* de 10, $m = 1.0$, $r = 1.0$. A ideia de setar $r = 1.0$ vem de que quanto mais indivíduos reproduzem, mais fácil será observar o efeito da pressão seletiva. Variamos β_{rank} de 1.0 até 2.0 com incrementos de 0.025



(a) Otimalidade



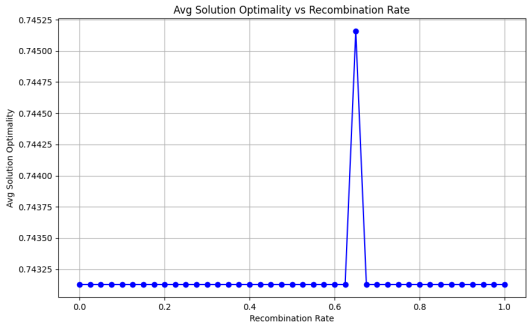
(b) Tempo

Figure 2: Pressão Seletiva

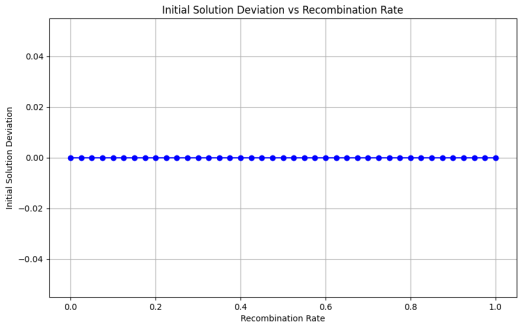
Como é possível observar, os resultados estão bastante ruidosos. Isso se deve ao fato de estarmos usando apenas 2 instâncias. Apesar disso, podemos observar um leve crescimento da otimalidade com um aumento de β_{rank}. Não é possível observar qualquer correlação com o tempo, o que faz sentido pois um aumento de β_{rank} apenas muda as probabilidades de seleção que são passadas para o PRNG.

6.3 *recombination_rate*

Realizamos dois testes. Primeiro utilizamos *population_size* de 50, *max_non_improving_generations* de 10, $m = 0.0$, $\beta_{rank} = 2.0$. Variamos r de 0.0 até 1.0 com incrementos de 0.025



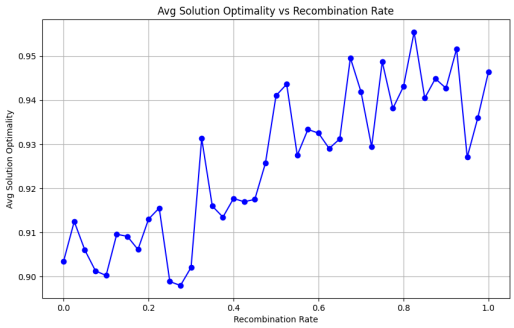
(a) Otimalidade



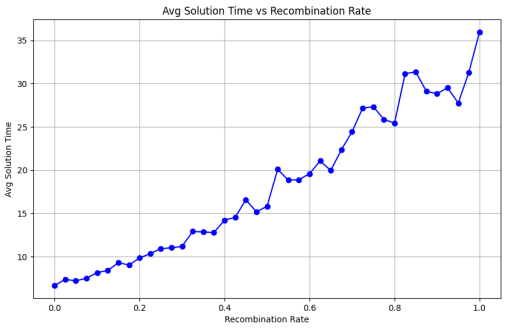
(b) Desvio SI na melhor semente

Figure 3: Taxa de recombinação $m = 0.0$

É possível observar que apenas em um dos testes a recombinação gerou um indivíduo melhor que a melhor solução inicial. Portanto, realizamos o mesmo teste, porém com $m = 1.0$



(a) Otimalidade



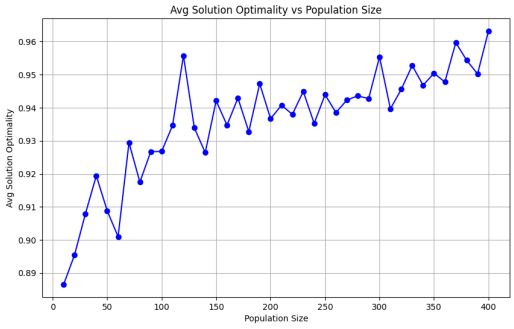
(b) Tempo

Figure 4: Taxa de recombinação $m = 1.0$

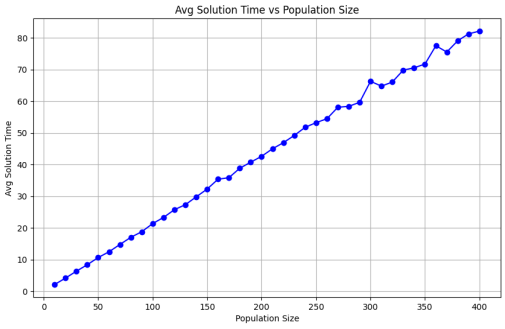
Tanto a otimalidade quanto o tempo crescem razoavelmente linearmente, apesar da otimalidade estar bastante ruidosa e parecer estagnar perto de 1.0. O que esses resultados indicam é que a recombinação introduz diversidade genética, a partir da qual a nossa implementação da mutação pode melhorar as soluções.

6.4 *population_size*

Utilizamos *max_generations* de 10, $r = 0.8$, $m = 0.45$, $\beta_{rank} = 2.0$. Variamos p de 10 até 400 com incrementos de 10.



(a) Otimalidade



(b) Tempo

Figure 5: Tamanho da população

Podemos observar um crescimento logarítmico da otimalidade, comparado com um crescimento linear do tempo. Como fixamos o número de gerações, podemos ter uma ideia do tempo por geração para instâncias com 500 ingredientes. Extrapolando para instâncias maiores, uma população muito grande reduz muito o número de gerações totais possíveis em 30min (ver seção 7).

7 Resultados Algorítmo Genético

Utilizamos $p = 100$, $r = 0.8$, $\beta_{rank} = 2.0$, $m = 0.45$ e um tempo máximo de 30 minutos. Como semente foi utilizado o tempo em nanossegundos de quando a instância foi executada.

Instância	BKV	SI	SF	Desvio SI	Desvio BKV	Tempo AG	Semente
ep01	2118	1955	2100	-7.42	0.85	1799.35	1724155016103110499
ep02	1378	966	1378	-42.65	0.00	1798.61	1724155016123311930
ep03	2850	2516	2850	-13.28	0.00	1798.16	1724155016107103687
ep04	2730	2494	2710	-8.66	0.73	1792.09	1724155016117448011
ep05	2624	2480	2619	-5.60	0.19	1796.97	1724155016123982456
ep06	4690	4300	4670	-8.60	0.43	1797.40	1724155016160802638
ep07	4440	4277	4420	-3.34	0.45	1798.14	1724155016238230331
ep08	5020	4359	5008	-14.89	0.24	1795.30	1724155016152963300
ep09	4568	4290	4540	-5.83	0.61	1773.65	1724156808443398996
ep10	4390	4242	4369	-2.99	0.48	1789.52	1724156811852088119

8 Resultados Solver GLPK

Executamos o programa inteiro formulado na seção 2 no solver GLPK versão 5.0, com um limite de tempo de 30 minutos.

Instância	BKV	Valor Solver	Desvio BKV
ep01	2118	1596	24.65
ep02	1378	957	30.55
ep03	2850	2748	3.58
ep04	2730	2620	4.03
ep05	2624	2510	4.34
ep06	4690	4554	2.9
ep07	4440	4332	2.4
ep08	5020	4929	1.8
ep09	4568	4450	2.59
ep10	4390	4241	3.4

Em todas as instâncias o valor obtido foi pior que o valor obtido via meta-heurística.

9 Conclusão

Consideramos que obtemos resultados satisfatórios, principalmente quando se compara com as soluções via solver. Recomendamos os parâmetros $p = 100$, $r = 0.8$, $\beta_{rank} = 2.0$, $m = 0.45$. Dito isso, reconhecemos que a nossa análise dos parâmetros teve resultados um tanto quanto duvidosos. O ideal teria sido gerar várias instâncias aleatórias pequenas (que talvez o solver consiga resolver) e usá-las para os testes. Como foi feito é como se fizéssemos um estudo com tamanho de amostra igual a 2. Ou seja, os parâmetros recomendados podem não ser os melhores de forma generalizada. Além disso, diversas otimizações podem ser feitas no código.

References

[1] John H. Holland. *Adaptation in Natural and Artificial Systems.* `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=b61cec42e5aa997a20d563b2886c083b3e0d335c`. Accessed: 2024-08-20. 1975.

[2] Ilan Schnell. *bitarray: Efficient arrays of booleans.* `https://pypi.org/project/bitarray/`. Version 2.7.4, Accessed: 2024-08-20. 2008.