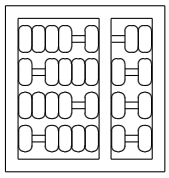

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Heurísticas para o
Problema de Precificação Livre de Inveja

Fabio Yudi Murakami

Rafael Crivellari Saliba Schouery

Technical Report - IC-16-09 - Relatório Técnico

December - 2016 - Dezembro

Heurísticas para o Problema de Precificação Livre de Inveja

Fabio Y. Murakami* Rafael C. S. Schouery†

Instituto de Computação, Universidade Estadual de Campinas, Campinas, Brasil

18 de dezembro de 2016

Resumo

Problemas de precificação visam determinar o preço de itens a serem vendidos com o objetivo de maximizar o lucro do vendedor. Neste trabalho propomos uma nova heurística para o problema de precificação livre de inveja com demanda unitária. Nesse problema temos como entrada os valores que os consumidores dão para os itens, e como saída uma precificação, isto é, um preço para cada item, bem como a alocação de itens à consumidores. A metaheurística utilizada foi o *Biased Random-Key Genetic Algorithm* (BRKGA) que é um algoritmo genético. Além disso, duas heurísticas previamente definidas na literatura, *Equal Price* e *Maximum Reservation Price*, foram utilizadas para gerar soluções a serem injetadas nas populações iniciais do algoritmo. A decodificação foi aprimorada utilizando uma rotina chamada *Improvement* que melhora o *fitness* de cada cromossomo. Ela faz isso encontrando os caminhos mínimos num grafo auxiliar. Esse grafo é construído com base na alocação encontrada a partir de uma precificação. De modo geral, o BRKGA conseguiu bons resultados em termos de qualidade das soluções e muitas vezes foi melhor e em outras ficou muito perto dos algoritmos da literatura, sendo melhor em aproximadamente 65% das instâncias testadas e ficando a pelo menos 97% do melhor valor nas instâncias restantes. Ele peca em relação ao tempo de execução, porém com alguns ajustes e pequenas otimizações foi possível melhorar tais tempos.

1 Introdução

O problema de precificação é comum na economia. Empresas enfrentam esse problema no mercado, já que necessitam escolher preços que otimizem seus ganhos. Preços escolhidos erroneamente podem ocasionar um lucro baixo para a empresa.

Problemas de precificação consistem em encontrar preços para os itens a serem vendidos de maneira a obter o maior lucro possível para o vendedor. A decisão de preços deve levar em conta o valor que cada comprador está disposto a pagar por um item. Com ele conseguimos ter uma ideia de "utilidade" do comprador olhando para a diferença entre

*email: ra138311@students.ic.unicamp.br
†email: rafael@ic.unicamp.br

o que ele estava disposto a pagar e o quanto ele realmente pagou. Podemos ver a utilidade como um lucro para o comprador, pois é o valor que ele economizou na compra do item.

Sendo assim, temos o caso específico do problema tratado nesse projeto que é o *problema de precificação de demanda unitária e livre de inveja*, ou *unit-demand envy-free pricing problem* (UEFPP). Ele é denominado de demanda unitária porque cada comprador está disposto a comprar no máximo uma cópia de qualquer item (considerando cada item com infinitas cópias), ou seja, ele comprará no máximo um item. E livre de inveja porque ele considera que cada comprador escolherá o item que lhe traga mais satisfação, o que não o fará desejar ter comprado outro item.

Desta maneira, para formalizar o UEFPP denotamos o conjunto de *consumidores* por $B = \{1, .., m\}$, e o conjunto de *itens* por $I = \{1, .., n\}$. A *valoração* é uma matriz v indexada por $I \times B$, onde v_{ib} representa o *valor* que o consumidor b dá para o item i. Os preços são representados por um vetor p indexado em I, com $p_i \in \mathbb{R}_+$. Denotamos a alocação como uma matriz x, também indexada em $I \times B$, onde, para cada consumidor b, $x_{ib} = 1$ se o item i for alocado para ele, e 0 caso contrário. Como a demanda é unitária, existe no máximo um item i, tal que, $x_{ib} = 1$ para todo comprador b.

A *utilidade* de um consumidor b, denotada por u_b, é o valor que este consumidor avalia o item i recebido menos o preço do item, ou seja, $u_b = v_{ib} - p_i$, caso i seja alocado para b e, $u_b = 0$, se b não receber o item.

Com todas as variáveis definidas, conseguimos modelar o nosso problema. A entrada é o valor que cada consumidor dá para cada item, ou seja, a matriz v. O objetivo é encontrar preços p e uma alocação x, onde o *lucro* do vendedor é o máximo. O lucro é definido como a somatória de todos os preços dos produtos que foram alocados para algum consumidor, lembrando que um item pode aparecer mais de uma vez na somatória. Como essa versão do problema é "livre de inveja", não deve existir, para um consumidor b, um item i, tal que, $v_{ib} - p_i \geq u_b$.

2 Trabalhos correlatos

Na literatura, vários trabalhos descrevem modelos para esse problema, como em [1–4] que descrevem o problema de precificação geral. Em particular, em Rusmevichientong et al. [5], foi introduzido o modelo que leva em consideração as estimativa de valor dado pelos consumidores, através da matriz v.

O modelo sugerido por Aggarwal et al. [6], foi formalizado por Guruswani et al. [7], que definia o conceito "livre de inveja". Para o caso da demanda unitária, estudado nesse trabalho, Guruswani et al. [7] apresentou uma $(2 \log n)$-aproximação, onde n é o número de consumidores e provou que esse problema é APX-difícil. Mais tarde, Hartline e Koltun [8] apresentaram uma aproximação $(1 + \epsilon)$-aproximação para quando o número de itens é constante.

Shioda, Tunçel e Myklebust [10] apresentaram heurísticas e uma formulação de programação inteira mista para o problema mais geral descrito por Guruswani et al. [7]. Em [11], apresentaram novas heurísticas, implementações mais rápidas para as heurísticas desenvolvidas por Dobson e Kalish [12] e propuseram uma fórmula de programação inteira

mista para o problema.

As heurísticas apresentadas em [11] são as que serão comparadas neste trabalho, pois apresentam os melhores resultados se tratando do problema de precificação e serão descritos a seguir.

Primeiramente, há uma heurística chamada de *Maximum Reservation Price*, ou MaxR, que utiliza o maior valor dado para cada item e é utilizada para encontrar uma solução inicial para a heurística que propomos.

Outra heurística, que também usamos a ideia para o nosso algoritmo, é a proposta por Dobson e Kalish [12] que, dado uma alocação, consegue encontrar uma solução resolvendo o problema de caminhos mínimos e é chamada de DK88.

Dobson e Kalish [13] também propuseram outra heurística chamada de DK93, que itera por todos os itens e busca o valor que leva ao maior lucro considerando os valores dados pelo consumidores e deixando todos os outros itens com preços fixos. Em [11], ainda são citados o Fast-DK e o Global-DK que são melhorias de DK88 e DK93, respectivamente.

Em [11], ainda é feita uma generalização dos algoritmos de Dobson e Kalish [12, 13], chamadas de *Generalized Reassignment Heurístic* (GRH), que levam a duas novas heurísticas, GRH-SubTree e Cell-Pierce. Também, utilizando a ideia da heurística de Guruswani et al. [7], apresenta uma heurística que chama de Guru. E outra chamada de $\overline{\text{Guru}}$, que aplica melhoras à saída do Guru, procedimento semelhante ao *Improvement*.

3 Justificativa

Como já foi citado, o problema de precificação é importante na economia, pois visa atender às demandas do mercado conseguindo otimizar os ganhos das empresas.

É um problema com um valor prático muito grande, porém está na classe de problemas NP-difíceis e não está em APX, a menos que P=NP, isto é, provavelmente não admite um algoritmo de aproximação com razão constante, como provado em [7].

Conseguindo bons algoritmos para o UEFPP, também é possível resolver problemas relacionados, como casos específicos do *Network Pricing Problem*.

O *Network Pricing Problem* (NPP) consiste em encontrar os preços a serem dados a pedágios, de forma que maximize o lucro do administrador da rede de transporte de *commodities*. Nele, dado um grafo dirigido com custos associados nos arcos, um conjunto de demanda para cada *commoditie* consistindo em uma origem, destino e quantidade demandada, e um subconjunto de arcos que são pedagiados, o objetivo é encontrar os preços para os pedágios que maximize o lucro, considerando que as *commodities* buscarão o caminho de menor custo (considerando o custo dos arcos mais os pedágios a serem pagos) entre a origem e o destino.

Para definir o NPP, começamos com $G = (V, A)$ sendo um grafo direcionado com V o conjunto de vértices e A o conjunto de arcos. Consideramos um vetor c, indexado em A, como o custo de cada arco. Temos também um conjunto $A' \subseteq A$ que representa os arcos com pedágios e um vetor t, indexado em A', que representa as tarifas do pedágio. Por último temos o conjunto K de *commodities*, onde cada elemento é representado por $k = (o_k, d_k, n_k)$, com $o_k \in V$, $d_k \in V$ e $n_k \in \mathbb{Q}$, denotando o origem, destino e quantidade demandada para

a *commoditie* k, respectivamente. Sendo assim, para dados G, c, A', K, o objetivo é encontrar t que maximize os lucros, ou seja, maximizar:

$$\sum_{k \in K} \sum_{\substack{a \in A', \\ a \in sp(o_k, d_k)}} n_k \times t_a$$

onde $sp(u, w)$ é o caminho mínimo entre $v, w \in V$, onde o custo de um arco é igual a $c_a + t_a$ se $a \in A'$, e c_a, caso contrário.

Um caso particular do problema é o NPP com pedágios conectados, onde o conjunto de pedágios forma um caminho. Três variantes são propostas para esse caso. O *Basic NPP*, onde cada arco tem um preço, e a *commoditie* pagará pela soma de pedágios de arcos usados. O *General Complete Toll NPP*, que tem preços definidos para cada par de entrada e saída do caminho. E, por último, o *Constrained General Complete Toll NPP*, onde os preços são definidos como o anterior, porém com restrições de monotonicidade e a desigualdade triangular.

Em Heilporn et al. [16], foi demonstrado que era possível fazer uma conexão entre o problema NPP e o problema de precificação de demanda unitária e livre de inveja. Uma instância do General Complete Toll NPP pode ser transformada em uma instância do UEFPP em tempo polinomial de maneira que, cada solução viável no UEFPP, também é viável no General Complete Toll NPP de mesmo valor. Em Fernandes et al. [14] foi demonstrado que o inverso também é possível. Em particular, isso significa que um algoritmo de aproximação ou heurística para o UEFPP pode ser usado para resolver o *General Complete Toll NPP*, retendo a qualidade da solução encontrada.

Para reduzir uma instância do General Complete Toll NPP em uma instância do UEFPP, basta considerar que o conjunto de itens é o conjunto de pares entrada-saída dos vértices do caminho, o conjunto de consumidores é o conjunto de *commodities* que passarão pela rede e, para cada consumidor k e item a, $v_{ak} = c^k_{od} - c^k_a$, onde c^k_{od} é o custo de um caminho mínimo de o_k para d_k que não utiliza arcos de pedágio e c^k_a o custo da estrada a. Com essa redução, é possível resolver o General Complete Toll NPP através de um algoritmo para o Unit-demand Envy-Free Pricing Problem.

Sendo assim, conseguindo algoritmos para o UEFPP, podemos usá-los também para resolver o General Complete Toll NPP. Ademais, estratégias usadas para o UEFPP podem ser interessantes de serem consideradas para o NPP, suas variantes e outros problemas de precificação.

4 Objetivos

O uso de meta-heurística visa resolver de forma genérica problemas de otimização, como o Envy-Free Pricing Problem, para os quais não se conhece algoritmos eficientes para encontrar uma solução.

Com isso, o objetivo deste trabalho é encontrar novos algoritmos para resolver o UEFPP que sejam melhores em desempenho que os já existentes, seja em qualidade da solução ou em tempo de execução. Em particular, investigaremos o uso da metaheurística BRKGA para resolver o problema.

5 Desenvolvimento do trabalho

A heurística escolhida para resolver o UEFPP foi o BRKGA. Esta seção conta com uma breve descrição desse algoritmo.

BRKGA

Algoritmos genéticos são técnicas usadas em problema de otimização ou busca, para encontrar um valor de solução baseando-se em processos estudados na biologia evolutiva como a hereditariedade, mutação, seleção natural e *cross-over*.

Entre os principais componentes de um algoritmo genético estão a função objetivo, que é o valor que desejamos otimizar, seja maximizar ou minimizar; o indivíduo, normalmente chamado de *cromossomo*, que é a maneira de representar uma solução possível para o problema; a *seleção*, que é a maneira de escolher indivíduos que farão parte do processo de formação da próxima geração; e a *reprodução*, que é o processo de evolução de geração em si.

Um *algoritmo genético de chaves aleatórias*, ou RKGA (*Random-Key Genetic Algorithm*), introduzido por Bean [19], é um algoritmo genético onde os cromossomos são representados por vetores de n chaves aleatórias. Cada chave aleatória, por sua vez, é representada por um número real no intervalo $[0, 1]$.

Uma *população* é formada por p indivíduos, sendo cada um deles um cromossomo. Cada indivíduo é classificado através do seu *fitness*, ou seja, um valor da solução correspondente no caso de um problema de otimização. Para poder gerar o valor do *fitness*, é preciso ter um *decodificador* que possa mapear o cromossomo em uma solução do problema e através dessa solução, calcular o valor. No RKGA, o único passo que dependem diretamente do problema é o cálculo da função de *fitness* feito pelo decodificador.

Com os *fitness* computados, é possível dividir a população em um conjuntos de p_e indivíduos que representam as melhores soluções, chamados de elite, e outro com $p - p_e$, chamados de não-elite. A partir daí, são escolhidos os indivíduos para formar a próxima geração. Eles podem ser apenas transferidos para a próxima geração, sofrer uma mutação, que garante maior variabilidade para fugir de valores ótimos locais, e através do cruzamento de dois indivíduos.

O BRKGA (*Biased Random-Key Genetic Algorithm*), introduzido por Gonçalves e Resende [20], é uma variante do RKGA. Ele difere do RKGA na forma em que os pais são selecionados para o cruzamento e como é implementado o cruzamento. Ele é chamado de viesado (*biased*), por escolher sempre um dos pais como um membro da elite, o que não acontece no RKGA, pois os dois pais são escolhidos aleatoriamente de toda a população.

O processo de formação da próxima geração é feito da seguinte forma no BRKGA. Supondo que o algoritmo se encontra na k-ésima geração, os p_e elementos da elite são transferidos para a $(k+1)$-ésima geração. Há também p_m indivíduos aleatórios gerados que compõem a próxima geração. Por fim, os últimos $p - (p_e + p_m)$ indivíduos que faltam para completar a geração vem do cruzamento de outros dois indivíduos. Um deles é escolhido aleatoriamente do grupo de elite e o outro, também aleatoriamente do grupo não-elite.

No *cruzamento*, ou *cross-over*, do BRKGA, um indivíduo a vem da elite e o outro, b, não-elite, formando o descendente c. Considerando o índice $i = \{1, .., n\}$, o alelo $c[i]$ do cromossomo é escolhido de $a[i]$ com uma probabilidade p, ou de $b[i]$ com probabilidade $1 - p$. Como o BRKGA é viesado em relação à elite, a probabilidade p é sempre maior que 0,5. Dessa forma, além de ter sempre um dos pais como membro da elite, existe uma maior chance do alelo do pai da elite ser repassado. Vale ressaltar que os pais sofrem reposição após o cruzamento, ou seja, eles podem ser escolhidos mais de uma vez para gerar um descendente.

Depois de realizados os $p - (p_e + p_m)$ cruzamentos, a nova geração está formada e começa-se uma nova iteração após a checagem dos critérios de parada. Eles podem ser os mais diversos, desde um número específicos de gerações, quanto a não melhora do resultado por um certo período de tempo ou até mesmo um determinado valor de solução.

Decodificador

Como citado anteriormente, a fase do BRKGA que depende do problema é apenas a decodificação para o cálculo da função de *fitness*. O decodificador deve mapear um cromossomo para uma solução do problema.

Em nosso algoritmo, cada cromossomo tem o tamanho n, que é o número de itens que estão à venda. Sendo assim, cada chave aleatória é mapeada para o preço de um item. Sem perda de generalidade, o preço p_i de cada item $i \in I$ é menor que o maior valor de v_{ib}. Em outras palavras, $p_i \leq \max\{v_{i'b} : i' \in I, b \in B\}$

Cada chave aleatória é multiplicada pelo maior valor que um comprador dá para um item. Dessa maneira, é como se cada alelo fosse uma fração do maior valor e assim todos os preços ficam dentro da faixa de interesse.

Com os preços é possível calcular o valor da função de *fitness*. Para isso, é preciso fazer a alocação dos itens para cada comprador. A alocação é feita usando o conceito de utilidade. Para cada comprador é alocado o item que lhe apresentar a maior utilidade para que a solução seja livre de inveja. Caso haja dois ou mais itens com utilidade máxima, a escolha é feita pelo item que dê maior lucro ao vendedor, ou seja, o que tiver o preço mais alto. Vale lembrar que nesse problema consideramos um estoque infinito de cada item e, por isso, um item pode ser alocado para qualquer número de compradores.

Depois de saber qual item foi alocado para cada comprador, basta somar os preços para obter o valor do lucro total. Esse valor é usado como *fitness* para poder selecionar os elementos do conjunto de elite da população e dessa forma dar prosseguimento ao algoritmo.

Algoritmo 1: Solution Value

```
   input : p, I, B, v
 1 totalProfit ← 0.0;
 2 foreach b in B do
 3 |   profit ← 0.0;
 4 |   utility ← 0.0;
 5 |   allocation[b] ← −1;
 6 |   foreach i in I do
 7 |   |   price ← p[i] ;
 8 |   |   if v_ib − p > utility or (v_ib − p = utility and profit < price) then
 9 |   |   |   utility ← value − p;
10 |   |   |   profit ← price;
11 |   |   |   allocation[b] ← i;
12 |   totalProfit ← totalProfit + profit;
13 return totalProfit
```

Na linha 2, o algoritmo percorre todos os consumidores do conjunto B, e inicializa o lucro, a utilidade e a alocação para o consumidor corrente nas linhas 3 a 5. Depois, na linha 6, percorrerá todos os itens em I verificando qual deles proporcionará a maior utilidade. Esse verificação é feita na linha 8, onde ele compara se o valor da utilidade do item é maior que a atual ou, se for igual, se tem o preço maior, já que o objetivo é maximizar o lucro do vendedor. Ao atualizar a utilidade e o lucro individual nas linhas 9 e 10, também atualiza a alocação para esse consumidor na linha 11. Na linha 12, soma-se o lucro individual do item ao lucro total do conjunto.

Além disso, duas soluções específicas são injetadas às populações iniciais do algoritmo para promover uma maior rapidez na convergência para um resultado melhor. Essas soluções são geradas através de duas heurísticas diferentes. Uma delas é o *Maximum Reservation Price Heurístic*, ou simplesmente *MaxR*, que é a escolha do preço de cada produto i com o maior valor dado dentre todos os consumidores, mais formalmente: $p_i = \max\{v_{ib} : b \in B\}$.

Conforme o algoritmo abaixo, percorrendo todos os itens (linha 1), verifica-se todas os valores avaliados pelos consumidores e escolhe-se o maior (linhas 3 a 5). Depois disso, armazena-se no vetor de preços p (linha 6).

Algoritmo 2: Maximum Reservation Price Heurístic

```
   input : p, I, B, v
 1 foreach i in I do
 2 |   maxPrice ← 0.0;
 3 |   foreach b in B do
 4 |   |   if maxPrice < v_ib then
 5 |   |   |   maxPrice ← v_ib;
 6 |   p[i] ← maxPrice;
 7 return p
```

A outra heurística é o *Equal Price Heurístic*, que, como o próprio nome sugere,

associa o mesmo preço para todos os itens, porém escolhe como preço o valor que dará o maior lucro. Em outras palavras, a heurística consiste em percorrer todos os valores de todos os consumidores a todos os itens e associar tal valor como preço de todos os itens. Depois de avaliar todos os itens, escolhe-se o valor que dê o maior lucro.

Algoritmo 3: Equal Price Heurístic

```
   input  : p, I, B, v
 1 maxProfit ← 0.0;
 2 maxPrice ← 0.0;
 3 foreach value in v_ib do
 4 |   foreach i in I do
 5 |   |   pricing[i] = value;
 6 |   profit ← SolutionValue(pricing, I, B, v);
 7 |   if profit > maxProfit then
 8 |   |   maxProfit ← profit;
 9 |   |   maxPrice ← value;
10 foreach i in I do
11 |   p[i] ← maxPrice
12 return p
```

Depois de geradas as duas soluções iniciais através do *Equal Price Heurístic* e *MaxR*, elas são injetadas no algoritmo BRKGA em populações diferentes, uma na primeira e outra na segunda população, respectivamente.

Também há um algoritmo chamado de *Improvement*, que dada uma alocação, encontra a melhor precificação para esta alocação. Com essa nova precificação, é possível achar uma nova alocação e repetir o processo até que não se consiga mais melhorar.

Para encontrar tal precificação, a rotina usa o dual do *LP* (*Linear Program*) que resolve o problema de encontrar uma precificação de lucro máximo para uma dada alocação. Tal dual corresponde à solução do problema dos caminhos mínimos no grafo de $n+1$ vértices, onde há um vértice para cada item e mais um vértice chamado de *dummy*. O comprimento do arco de um item i para outro j equivale ao mínimo da diferença, para cada consumidor b que recebe i, do valor de b para i e o valor de b para k. Formalmente, o comprimento do arco (i,j) é $\min\{v_{ib} - v_{jb} : b \in B, x_{ib} = 1\}$, onde x é a alocação dada. Para o *dummy*, o valor do arco para cada item j é o mínimo do valor dentre todos os consumidores que foram alocados para o item, isto é, $\min\{v_{ib} : b \in B, x_{ib} = 1\}$. Mais detalhes em [10, 12]

O novo preço de cada item será o equivalente ao comprimento do menor caminho saindo do nó *dummy* até o item.

A seguir encontra-se o algoritmo usado para o *Improvement*. Nele consideramos um grafo completo com $n + 1$ vértices, um mapa de arcos, *length*, com o valor do custo de cada um, inicializado com ∞. Além disso, o algoritmo de *Dijkstra()* recebe o mapa de distâncias e um vértice origem e retorna um vetor com a distância do caminho mínimo da origem para cada vértice.

Algoritmo 4: Improvement

```
input : p, allocation, v
foreach b in B do
    if allocation[b] < 0 then
        continue;
    j ← allocation[b];
    foreach i in I do
        if i ≠ j then
            if length[i, j] > v_ib − v_jb then
                length[i, j] ← v_ib − v_jb;
    if length[dummy, j] > v_ib then
        length[dummy, j] ← v_ib;
dist ← Dijkstra(length, dummy);
foreach i in I do
    new_p[i] ← dist(i);
return new_p
```

6 Testes computacionais

Para os testes de desempenho, foi utilizada uma máquina do Laboratório de Otimização Combinatório (LOCo) do Instituto de Computação da UNICAMP. A máquina rodava no Linux Ubuntu 12.04, com processador Intel(R) Core(TM) i7-2600 de 8 cores 3.4 GHz e 8GB de RAM.

Foram usadas as instâncias geradas pelo gerador descrito em [14]. Nele, são considerados quatro tipo de instâncias: *Characteristic*, *Neighborhood*, *Popularity* e *Random complete (RC)*, descritos a seguir.

Sendo assim, temos os seguintes tipos, descritos a seguir:

Characteristic

Esse modelo considera que um consumidor está interessado em um certo conjunto de características e terá interesse em qualquer item que tiver tais características. Por exemplo, em carros, um consumidor pode estar interessado em carros de uma certa cor e uma certa motorização. Sendo assim, esse consumidor potencialmente comprará qualquer carro que for dessa determinada cor e tiver a motorização desejada.

Neighborhood

No modelo *Neighborhood*, considera-se que itens e consumidores são pontos no plano e cada consumidor tem interesse em itens que estão próximos de sua localização no plano. Por exemplo, um consumidor que procura casas e tem uma certa região da cidade como favorita. Ele dará maior valor para as casas que se encontram perto dessa região.

Popularity

Considerando em mercado onde os produtos são parecidos, a preferência de um consumidor pode ser baseada na popularidade de um item específico. Esse modelo considera que consumidores estarão mais interessados em itens que forem mais populares. A popularidade de um item é medida pelo número de consumidores interessados nele, isto é, quanto maior o número de consumidores interessados, mais popular é o item. Sendo assim, um item popular tornará-se cada vez mais popular durante a geração da instância. Também é considerado que o preço é diretamente proporcional à qualidade do item e inversamente proporcional à sua popularidade. Como exemplo, podemos citar o mercado de *smartphones*, que são muito similares e a preferência do consumidor muitas vezes é baseada na popularidade do modelo.

Random Complete Instancies

Esse modelo é representado por um grafo bipartido completo, pois considera que cada consumidor $b \in B$ dá um valor para cada item $i \in I$. Esses valores são escolhidos aleatoriamente dentre um intervalo de valor mínimo e máximo considerado no problema.

Para os tipos Characteristic, Neighborhood e Popularity foram utilizadas instâncias de tamanho 50, 100, 150, 200, 250 e 300. Já para o tipo RC, apenas 100, 150, 200 e 250. O tamanho indica o número de consumidores, que é igual ao número de itens para essas instâncias.

Como mencionado, os parâmetros do BRKGA podem ser configurados de maneira flexível. Nossa implementação considerou os seguintes parâmetros: n e p, tamanho do cromossomo e da população, respectivamente, igual ao número de itens; fração da população considerada elite, $p_e = 0{,}2$; fração da população que será substituída por novos indivíduos aleatórios, $p_m = 0{,}1$; a probabilidade de um alelo do pai pertencente à elite ser passado para o descendente, $rhoe = 0{,}7$; números de populações independentes, $K = 3$; número máximo de gerações, MAX_GEN = 500; porcentagem de melhores indivíduos que serão substituídos X_NUMBER = $p/50$ a cada X_INTVL = 100 gerações; e por último, o número mínimo de gerações MIN_GEN = 150.

Para a rotina de *Improvement* foi usada a biblioteca de C++, LEMON [17], que oferece implementações eficientes para estruturas utilizadas em problemas de redes, como por exemplo grafos e o algoritmo de *Dijkstra*. Além disso, também utilizamos o código de API introduzido por Toso e Resende [21], para facilitar a implementação do BRKGA.

7 Resultados

A princípio, foram estudados os impactos da injeção de soluções dadas pelos algoritmos *MaxR* e *Equal Price* como indivíduos iniciais, neste momento ainda sem *Improve-*

ments. Os gráficos a seguir ilustram a execução de algumas instâncias, nelas percebemos que com as soluções iniciais, o BRKGA tem valores muitos melhores já nas primeiras gerações, o que o leva a convergir para valores melhores. Na Tabela 1 a seguir estão as heurísticas iniciais, juntamente com seu ganho médio por tipo de instância a partir do valor sem nenhuma heurística.

	EqualPricing	MaxR	Ambos
RC	34%	31,7%	34,4%
Neighborhood	1%	5%	5%
Popularity	1,5%	3%	3,5%
Characteristic	0,25%	0,5%	0,5%

Tabela 1: Ganho médio de cada heurística inicial comparado com o algoritmo sem nenhuma delas.

Pode-se notar nos gráficos das Figuras 1 a 4 que, no geral, utilizando-se as duas heurísticas, *MaxR* e *EqualPricing*, temos valores iniciais melhores. O *EqualPricing* apresenta valores melhores para os tipos *Characteristic* e *RC*, já o *MaxR* apresenta melhores soluções para os tipo *Neighborhood* e *Popularity*. Usando as duas heurísticas, conseguimos sempre o melhor valor. Por exemplo, na Figura 1, o valor sem heurísticas na primeira geração é 3575,1. Esse valor aumenta para 4130,57 usando *MaxR*, 6369,35 usando *EqualPricing* e os mesmos 6369,35 usando ambos. Já na Figura 2, o valor sem heurísticas é 19027,7, com *MaxR* é 47523,5, com *EqualPricing* é 31005,5 e 47523,5 com ambos.

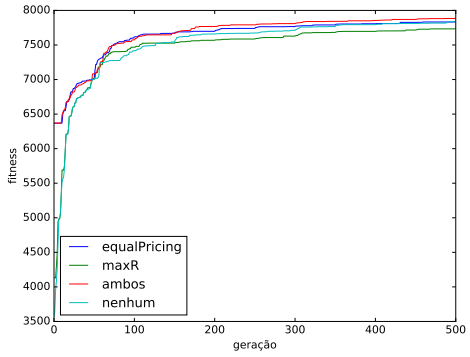

Figura 1: Gráfico *fitness* x geração da execução de instância tipo *Characteristic*.

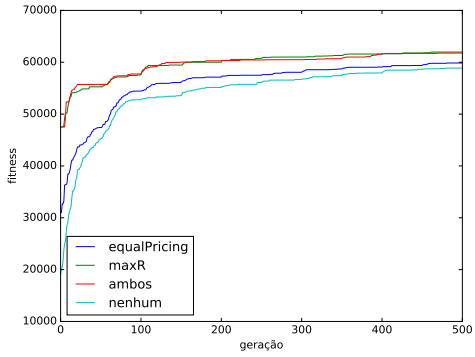

Figura 2: Gráfico *fitness* x geração da execução de instância tipo *Neighborhood*.

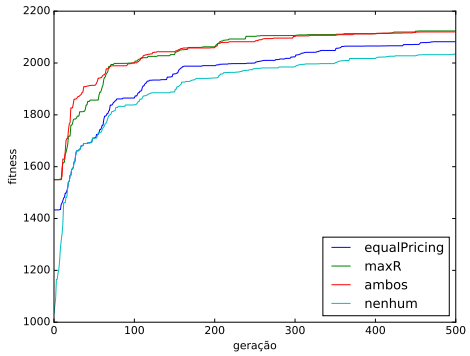

Figura 3: Gráfico *fitness* x geração da execução de instância tipo *Popularity*.

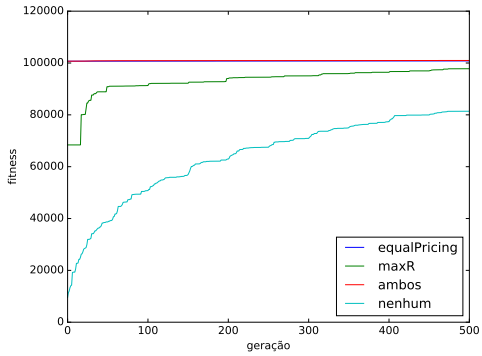

Figura 4: Gráfico *fitness* x geração da execução de instância tipo *RC*.

Com esses resultados, optamos por usar as duas heurísticas para gerar soluções iniciais para o nosso algoritmo.

Nas tabelas a seguir estão os valores obtidos, juntamente com o tempo de execução. Pode-se notar que existem três versões do BRKGA com *Improvement* e uma sem. O *OneImprove* quer dizer que o algoritmo de *Improvement* roda apenas uma vez tentando melhorar o resultado. No *TwoImprove*, indica que são feitas duas tentativas de *Improvement* em cada decodificação. Por último, o *MaxImprove*, é a versão em que o algoritmo é rodado até não conseguir melhorar mais o resultado, ou seja, o maior número de vezes. A versão sem *Improvement*, ou *NoImprove*, é basicamente o BRKGA normal, exceto pelos *Improvements* feitos nas soluções iniciais injetadas. Os algoritmos da literatura são o *maxR*, *guru*, $\overline{guru}$, $\overline{guru}$*GRHsubtree* (GRH-Subtree inicializado com $\overline{guru}$) na tabela abreviado como *gbGRHst*, *maxRnldk* (*Global-DK* inicializado com *maxR*) e *maxRpc* (*maxR* com *price change*).

tipo	n	cellpierce	guru	$\overline{\text{guru}}$	gbGRHst	maxRnldk	maxRpc	maxr	noImprove	oneImprove	twoImprove	maxImprove
c.0	50	4119.60	3595.32	3981.54	4108.82	4108.82	4045.11	3538.05	2445.97	**4214.73**	**4214.73**	**4214.73**
c.4	50	3940.26	3345.78	3795.20	3940.26	3940.26	3940.26	3511.24	2451.02	**4014.57**	**4014.57**	**4014.57**
c.0	100	7900.53	6369.35	7393.10	7721.61	7833.60	7884.31	7238.84	7653.17	8107.53	8086.04	**8122.37**
c.3	100	8374.73	6586.44	7991.86	8282.67	8282.67	8393.80	7991.86	8113.35	**8704.86**	8694.54	8693.88
c.9	100	8456.53	6757.66	8154.90	8342.70	8379.12	8496.13	7643.57	8094.84	**8776.98**	8774.54	8776.27
c.0	150	12439.10	9896.22	11863.87	12169.93	12308.25	12373.43	11538.24	12165.00	12819.10	**12914.30**	12838.40
c.4	150	11906.07	8954.65	10611.94	11152.10	11572.74	11830.04	9397.21	11483.80	**11964.30**	11960.50	11956.50
c.0	200	15682.53	12142.73	14888.67	15400.95	15442.75	15585.05	14467.21	15086.90	15945.30	**16022.40**	15900.20
c.2	200	15879.42	12650.14	14819.58	15267.62	15670.32	15687.69	12057.57	15068.00	15986.50	**15991.70**	15927.70
c.3	200	17480.14	13537.26	16006.36	17092.30	17249.11	17506.71	15678.86	16670.10	17831.60	17801.60	**17859.30**
c.6	250	20543.09	15919.04	18735.53	19769.72	20374.56	20357.20	17575.17	19421.40	20675.90	20689.10	**20715.40**
c.3	250	21285.85	16445.61	19728.26	20155.49	21065.94	21192.21	18862.62	20134.10	**21563.50**	21474.10	21421.50
c.0	300	24641.14	19605.99	23275.59	24160.79	24427.32	24414.76	23126.75	23543.70	24915.70	**25051.10**	24899.20
c.2	300	23998.90	18524.13	22551.25	22809.12	23806.22	24023.15	22180.18	22704.10	24196.40	24267.20	**24427.10**
c.8	300	24469.38	19088.71	22718.51	23821.14	24090.76	24368.54	21552.45	23118.90	24782.20	24696.70	**24799.50**
p.0	50	1091.09	766.18	1041.28	1079.82	1086.52	1085.33	1019.43	1063.99	1105.77	1106.63	**1108.49**
p.1	50	984.58	748.34	963.58	983.98	1012.79	921.20	884.69	989.07	1040.95	**1043.10**	**1043.10**
p.0	100	2099.96	1433.23	1809.46	1877.35	2021.21	1969.66	1416.15	1970.18	2192.39	2195.89	**2204.25**
p.3	100	2066.83	1495.89	1868.33	1894.92	1966.78	1959.19	1376.52	1894.92	2085.28	**2095.08**	2094.81
p.4	100	2073.11	1432.93	1873.95	1873.95	2005.50	2071.84	867.85	1927.41	2112.12	2117.82	**2129.97**
p.0	150	3021.48	2158.41	2764.08	2764.08	2954.23	2973.74	2657.13	2744.70	3092.23	3108.17	**3120.53**
p.1	150	3129.00	2310.52	2803.76	2817.89	2982.57	3122.90	2644.40	2927.22	3207.27	**3263.13**	3210.06
p.0	200	3755.66	2742.55	3628.12	3745.33	3781.30	3789.48	3150.99	3431.22	**3929.42**	3880.80	3905.13
p.2	200	3971.97	3026.20	3720.54	3822.51	3975.18	4006.25	3348.07	3743.34	4108.49	4136.55	**4146.75**
p.4	200	3941.91	2842.46	3667.61	3667.61	3867.81	4028.85	3325.12	3644.83	**4085.26**	4063.01	4052.61
p.0	300	5489.43	4244.54	5126.29	5173.32	5547.61	5656.82	4622.75	5126.86	**5787.34**	5742.58	5717.57
p.2	300	5906.38	4178.34	5316.79	5316.79	5732.91	5863.30	4503.81	5192.08	5875.68	5961.92	**5992.90**
p.8	300	5862.51	4283.84	5278.27	5395.62	5765.66	5746.24	4785.16	5351.43	**5959.33**	5872.67	5872.56

Tabela 2: Valores da solução de todos os algoritmos testados - Tipo Characteristic e Popularity.

tipo	n	cellpierce	guru	$\overline{\text{guru}}$	gbGRHst	maxRnldk	maxRpc	maxr	noImprove	oneImprove	twoImprove	maxImprove
n.0	50	**13573.87**	5696.44	10389.50	12937.36	13555.82	13108.50	7748.30	2232.95	9982.90	9890.09	10507.80
n.2	50	20207.86	7130.88	18931.05	19187.71	**20214.53**	19501.73	8037.84	2179.94	12130.50	16835.00	17331.00
n.0	100	**41577.70**	17357.42	37211.49	38194.95	41468.64	40656.49	25925.43	8678.42	38733.00	37899.10	38260.70
n.2	100	**54504.84**	20858.14	42467.87	49534.34	52488.21	52749.67	29701.86	8536.87	40846.00	46660.80	46964.00
n.3	100	30366.23	14753.06	28567.31	28875.78	**30742.67**	28724.78	19988.58	8746.89	26447.40	26112.20	26103.70
n.0	150	**65012.21**	31005.55	55450.87	59348.39	64512.64	64332.23	41930.79	19118.10	55186.20	56926.50	58045.20
n.1	150	**77096.05**	31084.84	69671.38	69671.38	76803.79	73123.41	44598.99	18675.60	68000.10	67257.20	68459.40
n.0	200	**105880.89**	42023.48	92622.60	92622.60	103242.48	98895.65	58947.99	31629.90	94731.80	101197.00	98068.00
n.1	200	104564.32	46616.40	92774.59	92774.59	**105648.06**	102487.19	61515.14	33954.60	95838.10	100076.00	99015.20
n.0	250	**136398.94**	62237.25	123031.77	123031.77	135319.46	123069.33	66020.94	47019.50	126418.00	124820.00	126263.00
n.5	250	**149587.21**	56049.21	126906.79	126906.79	147884.58	138545.28	91272.97	46963.50	135434.00	136777.00	137894.00
n.6	250	**145503.76**	62714.14	123536.07	123536.07	143680.99	142960.41	96643.17	52084.80	128411.00	131715.00	128649.00
n.0	300	**205002.24**	82574.90	178346.95	178346.95	204049.83	197768.24	124756.49	67253.80	185284.00	190637.00	191708.00
n.2	300	182162.40	78533.78	160352.92	160352.92	**182437.64**	177665.22	111194.87	69080.40	169712.00	173141.00	172539.00
n.8	300	**185101.16**	77430.98	164963.39	164963.39	183145.12	174362.68	103302.57	67643.30	168955.00	172203.00	169220.00
rc.0	100	101704.00	100700.00	101584.00	101704.00	101708.00	101682.00	101584.00	9991.52	101704.00	**101718.00**	101714.00
rc.1	100	101115.00	98703.00	100211.00	101013.00	100389.00	101204.00	99382.00	9983.68	101204.00	101199.00	**101231.00**
rc.3	100	101615.00	100700.00	101597.00	101610.00	101615.00	101610.00	101597.00	9981.58	101625.00	**101636.00**	101620.00
rc.0	150	152623.00	150900.00	152184.00	152368.00	152620.00	**152625.00**	152184.00	22478.20	152426.00	152574.00	152525.00
rc.3	150	152611.00	150341.00	151582.00	152097.00	**152623.00**	152603.00	151582.00	22487.60	152511.00	152495.00	152552.00
rc.4	150	152389.00	150300.00	151912.00	152180.00	**152394.00**	152388.00	151912.00	22467.60	152149.00	152149.00	152149.00
rc.0	200	**203843.00**	202600.00	203694.00	203790.00	203833.00	203832.00	203694.00	39982.50	203750.00	203753.00	203753.00
rc.2	200	203873.00	201400.00	203635.00	203704.00	**203879.00**	203875.00	203635.00	39970.00	203527.00	203561.00	203561.00
rc.8	200	**203774.00**	201600.00	203454.00	203537.00	203764.00	203759.00	203454.00	39980.70	203554.00	203554.00	203554.00
rc.0	250	254815.00	252500.00	254521.00	254651.00	**254843.00**	254832.00	254521.00	62477.50	254546.00	254546.00	254546.00
rc.2	250	**255006.00**	252500.00	254825.00	254918.00	255004.00	255000.00	254825.00	62480.30	254717.00	254723.00	254723.00
rc.4	250	254933.00	252750.00	254637.00	254774.00	**254936.00**	254925.00	254637.00	62472.60	254565.00	254587.00	254587.00

Tabela 3: Valores da solução de todos os algoritmos testados - Tipo Neighborhood e RC.

tipo	n	cellpierce	guru	$\overline{\text{guru}}$	gbGRHst	maxRnldk	maxRpc	maxr	noImprove	oneImprove	twoImprove	maxImprove
c.0	50	0.04	0.00	0.01	0.04	0.01	0.02	0.00	0.12	1.35	2.54	3.88
c.4	50	0.05	0.00	0.01	0.05	0.02	0.02	0.00	0.13	1.34	2.55	3.50
c.0	100	0.34	0.02	0.09	0.28	0.13	0.14	0.01	0.60	5.53	10.32	19.60
c.3	100	0.26	0.02	0.09	0.28	0.07	0.12	0.00	0.62	5.52	10.41	17.50
c.9	100	0.19	0.01	0.09	0.32	0.07	0.14	0.00	0.60	5.50	10.37	17.33
c.0	150	1.13	0.06	0.30	1.66	0.34	0.48	0.00	1.34	12.37	23.51	44.48
c.4	150	0.93	0.06	0.30	0.89	0.50	1.06	0.00	1.29	12.08	22.61	44.91
c.0	200	1.22	0.14	0.70	3.67	1.36	1.39	0.01	2.26	20.17	39.84	74.80
c.2	200	1.72	0.14	0.70	3.86	1.58	1.52	0.01	2.26	20.20	37.87	74.69
c.3	200	1.70	0.14	0.70	5.53	1.61	1.47	0.00	2.28	20.36	38.13	85.67
c.6	250	4.74	0.27	1.36	6.42	3.01	2.53	0.01	3.68	35.45	66.77	145.71
c.3	250	2.24	0.27	1.37	3.30	2.97	2.93	0.01	3.60	35.37	68.78	138.78
c.0	300	3.33	0.47	2.35	15.35	5.41	3.62	0.01	5.18	49.83	91.01	180.66
c.2	300	2.73	0.47	2.36	4.57	5.62	4.40	0.02	5.19	48.11	92.34	184.25
c.8	300	5.00	0.47	2.34	17.53	4.99	4.13	0.01	5.12	48.15	90.10	197.56
p.0	50	0.07	0.01	0.01	0.06	0.02	0.02	0.00	0.13	1.38	2.68	3.65
p.1	50	0.04	0.00	0.01	0.02	0.02	0.01	0.00	0.13	1.39	2.57	3.73
p.0	100	0.45	0.02	0.09	0.29	0.17	0.21	0.00	0.57	5.09	9.48	14.05
p.3	100	0.35	0.01	0.09	0.26	0.12	0.14	0.00	0.56	5.10	9.36	13.73
p.4	100	0.86	0.02	0.09	0.12	0.17	0.19	0.00	0.56	5.09	9.61	14.38
p.0	150	1.04	0.06	0.30	0.39	0.54	0.52	0.00	1.25	11.27	20.94	32.33
p.1	150	0.80	0.06	0.30	0.48	0.55	0.72	0.00	1.25	11.36	21.38	34.23
p.0	200	1.88	0.14	0.70	2.58	0.94	0.79	0.00	2.30	18.96	37.71	61.02
p.2	200	1.42	0.14	0.71	2.31	1.37	1.26	0.00	2.31	19.54	37.02	58.66
p.4	200	2.08	0.14	0.71	0.90	1.40	1.62	0.02	2.29	19.41	37.23	86.73
p.0	300	8.05	0.47	2.37	4.14	5.67	4.46	0.01	5.21	45.88	88.74	152.77
p.2	300	7.20	0.47	2.37	2.99	4.76	4.48	0.01	5.33	45.89	87.19	139.78
p.8	300	7.82	0.47	2.37	4.64	5.82	3.73	0.01	5.18	45.77	86.26	163.25

Tabela 4: Tempos de execução de todos os algoritmos testados - Tipo Characteristic e Popularity.

tipo	n	cellpierce	guru	$\overline{\text{guru}}$	gbGRHst	maxRnldk	maxRpc	maxr	noImprove	oneImprove	twoImprove	maxImprove
n.0	50	0.05	0.00	0.01	0.02	0.03	0.03	0.00	0.11	1.17	2.27	3.06
n.2	50	0.04	0.00	0.02	0.03	0.04	0.03	0.00	0.10	1.25	2.40	3.37
n.0	100	0.30	0.01	0.10	0.28	0.47	0.28	0.00	0.53	5.10	9.79	16.14
n.2	100	0.40	0.02	0.09	0.19	0.37	0.25	0.00	0.53	5.08	10.20	17.18
n.3	100	0.22	0.01	0.09	0.16	0.40	0.23	0.00	0.51	4.87	9.36	15.03
n.0	150	0.97	0.06	0.30	0.69	1.62	0.92	0.00	1.14	10.93	20.74	30.56
n.1	150	0.89	0.06	0.31	0.38	1.45	0.84	0.00	1.22	11.62	22.24	34.00
n.0	200	2.11	0.14	0.71	0.91	5.14	2.53	0.00	2.27	20.53	38.57	58.27
n.1	200	2.13	0.14	0.71	0.91	4.22	1.74	0.01	2.24	20.08	38.22	58.73
n.0	250	5.05	0.27	1.39	1.75	14.82	4.19	0.01	3.55	32.22	61.28	91.77
n.5	250	3.59	0.27	1.43	1.80	13.00	3.92	0.01	3.48	34.49	64.54	96.77
n.6	250	4.01	0.28	1.39	1.79	12.74	4.84	0.01	3.65	36.07	69.38	118.18
n.0	300	7.80	0.47	2.40	3.03	27.94	8.50	0.02	5.39	51.98	97.94	171.82
n.2	300	6.65	0.46	2.40	3.16	30.30	9.39	0.01	5.41	51.66	97.51	154.28
n.8	300	7.14	0.47	2.39	2.95	32.52	9.20	0.01	5.49	51.58	97.88	160.84
rc.0	100	0.52	0.02	0.09	0.57	0.19	0.11	0.00	5.99	56.42	107.31	206.30
rc.1	100	0.97	0.02	0.10	0.42	0.18	0.34	0.00	5.89	56.77	108.71	231.43
rc.3	100	0.18	0.02	0.10	0.20	0.07	0.03	0.00	5.98	56.73	108.29	211.39
rc.0	150	2.44	0.06	0.31	2.19	1.05	0.58	0.01	18.60	195.45	377.82	826.41
rc.3	150	2.59	0.06	0.31	2.16	0.91	0.64	0.00	18.88	197.16	372.44	780.83
rc.4	150	2.47	0.06	0.31	2.31	0.89	0.42	0.00	18.86	195.30	372.11	839.87
rc.0	200	5.08	0.14	0.74	4.54	2.77	1.20	0.01	44.28	480.37	920.89	2189.30
rc.2	200	5.31	0.14	0.73	5.35	1.91	0.90	0.00	44.79	479.12	911.17	2124.39
rc.8	200	6.28	0.14	0.72	5.79	2.44	0.96	0.00	44.31	477.00	905.62	2163.17
rc.0	250	13.85	0.27	1.42	13.83	8.37	2.82	0.01	83.65	926.67	1729.94	4185.28
rc.2	250	8.72	0.27	1.43	8.71	5.34	1.82	0.01	84.36	890.12	1718.56	4186.55
rc.4	250	13.09	0.28	1.42	13.06	6.90	2.36	0.01	84.97	900.77	1774.88	4141.38

Tabela 5: Tempos de execução de todos os algoritmos testados - Tipo Neighborhood e RC.

Também são apresentados alguns gráficos de execução de uma instância de cada tipo. Neles, o melhor valor de um algoritmo da literatura está representado com o nome do algoritmo que obteve tal resultado. O *NoImprove* não é considerado, pois, depois de analisar os primeiros resultados, notamos que sempre apresenta valores abaixo dos demais.

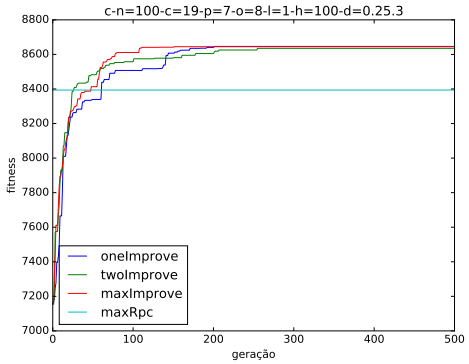

Figura 5: Gráfico *fitness* X geração em comparação com o melhor algoritmo - *Characteristic*.

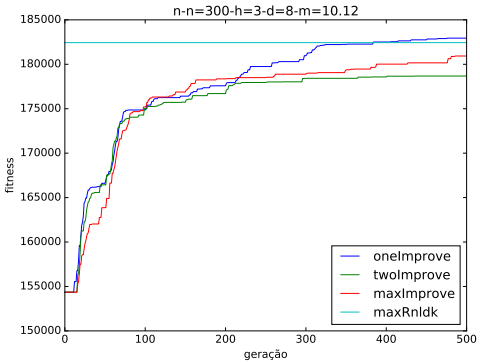

Figura 6: Gráfico *fitness* X geração em comparação com o melhor algoritmo - *Neighborhood*.

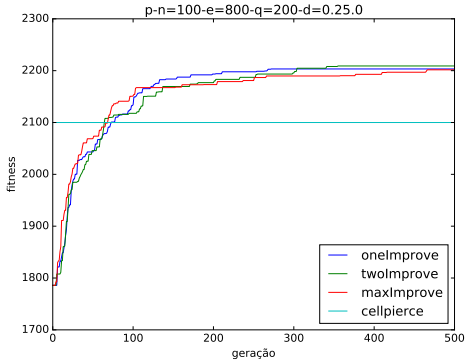

Figura 7: Gráfico *fitness* X geração em comparação com o melhor algoritmo - *Popularity*.

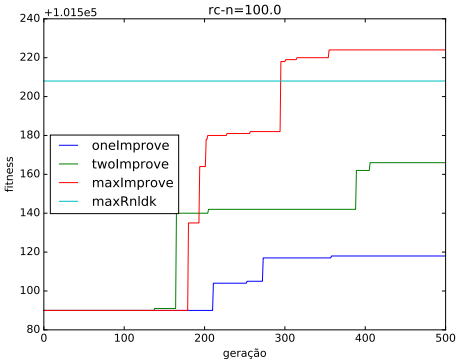

Figura 8: Gráfico *fitness* X geração em comparação com o melhor algoritmo - *RC*.

Na Figura 5, temos o *MaxImprove* com o melhor resultado, ficando com 8644,57 e ultrapassando o melhor valor de 8393,79 (*MaxRpc*) em apenas 48 gerações. Na Figura 6, vemos a dificuldade que o algoritmo tem nas instâncias de tipo *Neighborhood*, onde o *OneImprove* foi melhor por pouco, ficando com 182942,00 contra 182437,638 (*MaxRnldk*), uma diferença de apenas 0,2% e ultrapassando na geração 384. Na Figura 7, temos o *TwoImprove* como o melhor valor, 2209,06, contra 2099,955 (*Cellpierce*), ultrapassando na geração 65. Por último, na Figura 8 temos o *MaxImprove* com 101724,00 e o *MaxRnldk* com 101708,00,

ultrapassando em geração 295.

As versões do algoritmo BRKGA com *Improvement*, em várias instâncias, melhora o resultados dos algoritmos da literatura e em outras, fica muito perto do valor. Em particular, o desempenho do nosso algoritmo é melhor nas instâncias do tipo *Characteristic* e *Popularity* e não tão bem no tipo *Neighborhood*. Nos tipos *RC*, ele não bate o valor em todas as instâncias, mas fica muito próximo. Esses resultados são melhores apresentados em um gráfico de desempenho. Eles ilustram a frequência a qual o algoritmo obteve resultados próximos ao melhor e o quão perto ficou. Por exemplo, o *OneImprove* obteve 99% do melhor valor em $72,7\%$ das instâncias, enquanto foi o melhor em $52,7\%$

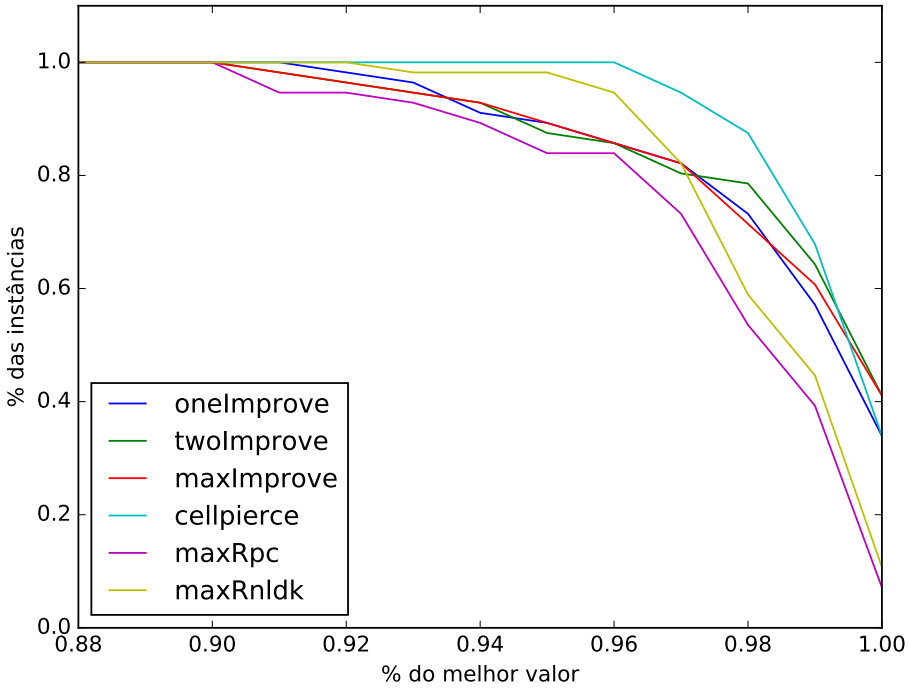

Figura 9: Gráfico de desempenho dos melhores algoritmos para todos os tipos de instâncias.

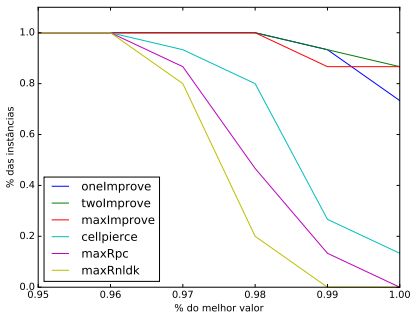

Figura 10: Gráfico de desempenho para instâncias tipo *Characteristic*.

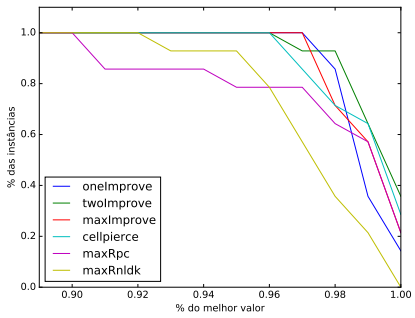

Figura 11: Gráfico de desempenho para instâncias tipo *Popularity*.

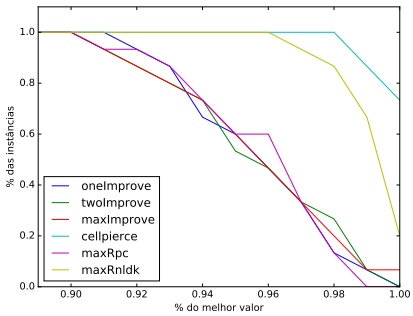

Figura 12: Gráfico de desempenho para instâncias tipo *Neighborhood*.

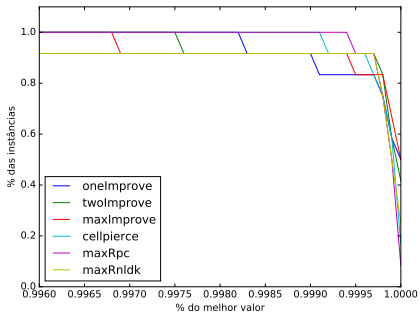

Figura 13: Gráfico de desempenho para instâncias tipo *RC*.

Nas Figuras 10, 11 e 13, pode-se notar que os algoritmos do BRKGA com *Improvement* têm melhor desempenho e que são os algoritmos que mais têm valores de melhor resultado de instâncias. Apenas nas instâncias de tipo *Neighborhood* (Figura 12) o melhor desempenho é do *Cellpierce*. No geral, o *TwoImprove* e o *MaxImprove* são os que apresentaram mais vezes os melhores valores para as instâncias. O *OneImprove* ficou empatado com o *Cellpierce* em segundo lugar.

Apesar dos valores superarem os outros algoritmos, o problema do BRGKA é o tempo de execução. Por exemplo, para a execução da instância do tipo *Characteristic* de tamanho 300, o melhor algoritmo da literatura (*Cellpierce*) dura um tempo de $5,1$ segundos, enquanto o *OneImprove* (mais rápido do BRKGA) roda em $48,15$ segundos, que equivale a um valor mais de 9 vezes maior. Por isso, utilizando-se o profiler *GNU Gprof*, foi feita a análise da execução de algumas instâncias. Ficou claro que o grande gargalo do algoritmo é a execução do *Improvement*, chegando a ocupar pouco mais de 92% de todo o tempo de execução.

Com isso em mente, o objetivo foi otimizar o máximo a implementação do método de *Improvement*. Dentro do método, o algoritmo de *Dijkstra* é a rotina que mais pesa na

execução, então, uma das ideias foi mudar o tipo de grafo usado para melhorar o desempenho do *Dijkstra*. Umas das hipóteses era que o *FullDigraph* da biblioteca *Lemon*, que é um grafo com todas os arcos possíveis, poderia deixar o algoritmo lento por ser muito denso. Sendo assim, os testes feitos foram a troca do *FullDigraph* para *ListDigraph* ou *SmartDigraph*. Os resultados na execução de algumas instâncias se encontram na tabela 6 seguir.

		FullDigraph	ListDigraph	SmartDigraph
Tipo	n	Tempo	Tempo	Tempo
Characteristic	100	14.4919	8.18316	**5.77337**
	150	45.4236	18.7041	**13.221**
	200	101.55	30.7643	**21.9446**
	250	187.888	53.757	**40.2105**
Neighborhood	100	14.2327	7.08677	**5.41413**
	150	43.5428	14.9483	**11.4198**
	200	99.292	26.3988	**20.5227**
	250	186.532	41.1264	**30.2065**
Popularity	100	14.4258	8.42597	**6.16668**
	150	44.7891	19.6071	**14.2477**
	200	102.637	34.5217	**25.2537**
	250	191.578	56.0363	**40.2195**
RC	100	**40.3679**	89.5368	61.6957
	150	**146.835**	354.001	227.899
	200	**436.801**	995.277	790.311

Tabela 6: Comparação de tempo de execução entre tipos de grafo para OneImprove, destacados os menores tempos.

Na tabela estão destacados os menores tempos de execução. Como se pode notar, a melhora chega, em alguns casos, à 70% com o *ListDigraph* e à 80% com o *SmartDigraph*. O único caso onde o *FullDigraph* funciona bem é nas instâncias do tipo *RC*, pois são instâncias que são densas.

Em [18] é descrito um algoritmo de Dijkstra ideal para grafos densos. Ele baseia-se no algoritmo de árvore geradora mínima de *Prim* para construir a árvore de caminhos mínimos do grafo que contém o valor da distância para cada vértice. Ele não possui uma fila de prioridades e, por isso, é ideal para grafos densos, pois as operações de alteração de prioridades se tornam significantes quando consideradas o número de vezes que serão efetuadas.

Na tabela a seguir estão os resultados comparando os valores do algoritmo inicial (com o *Heap* de prioridades) e a nova implementação. Na última coluna encontra-se o diferencial no tempo de execução de cada instância.

Instância	maxImprove	oneImprove	twoImprove
rc-n=100.0	53.4%	62.0%	57.8%
rc-n=100.1	52.8%	61.5%	57.2%
rc-n=100.3	52.9%	61.4%	57.0%
rc-n=150.0	55.8%	63.7%	59.5%
rc-n=150.3	53.1%	62.6%	58.6%
rc-n=150.4	54.6%	62.4%	58.3%
rc-n=200.0	52.9%	62.9%	58.9%
rc-n=200.2	52.7%	63.0%	59.5%
rc-n=200.8	53.7%	64.3%	59.9%
rc-n=250.0	52.6%	64.6%	59.8%
rc-n=250.2	52.5%	66.7%	62.4%
rc-n=250.4	53.4%	66.1%	61.0%

Tabela 7: Fração do tempo de execução do algoritmo de Dijkstra sem heap em relação à versão com heap.

Através dos resultados, vemos que o novo algoritmo baseado em [18] realmente é mais eficiente neste tipo de instância. Ele obteve de 52,5% até 66,7% do tempo em relação à versão com fila de prioridades.

Como a densidade é uma propriedade da instância que podemos calcular em tempo de execução, fizemos o nosso algoritmo distinguir entre as instâncias para que rode o Dijkstra do LEMON nas instâncias menos densas e o baseado em [18] nas mais densas.

8 Conclusões

Neste trabalho introduzimos o BRKGA (*Biased Random-Key Genetic Algorithm*) como metaheurística para solucionar o problema de precificação livre de inveja. Verificamos os benefícios de introduzir soluções nas populações iniciais do algoritmo com as heurísticas *Equal Price* e a *Maximum Reservation Price*. Também propusemos uma rotina que chamada de *Improvement* que melhorou o *fitness* que cada cromossomo na hora de decodificá-los, para tal, utiliza um grafo auxiliar formado pelos itens e resolve o problema de caminhos mínimos através do algoritmo de *Dijkstra*.

A implementação do *Dijkstra* em conjunto com a representação *SmartDigraph* de grafo direcionado do LEMON foi a que apresentou melhor desempenho nos modelos onde os grafos eram esparsos. Já nos grafos densos, o LEMON já não foi tão eficiente. Por isso, fizemos uma implementação do *Dijkstra* sem filas de prioridade, indicado para grafos densos, e obtivemos melhoras de 38% a 47% no tempo de execução.

Apesar de perder em tempo para os outros algoritmos, o BRKGA obteve a melhor solução em aproximadamente 66% das instâncias testadas. Nas demais, as soluções apresentadas ficaram 3% abaixo do melhor algoritmo.

Os melhores desempenhos foram nas instâncias de tipo *Characteristic* e *Popularity*, obtendo melhor valor em todos os teste, se rodados até o limite de gerações (apesar do maior

tempo). Utilizando novos critérios de parada, essas instâncias foram melhores em até 86%.

Sendo assim, podemos concluir que o BRKGA é um bom algoritmo para resolver o problema de precificação livre de inveja se houver necessidade de melhores soluções, e disponibilidade de tempo.

Referências

[1] S.S. Oren, S.A. Smith, R.B. Wilson, Pricing a product line, J. Bus. 57 (1) (1984) S73–S79.

[2] S.S. Oren, S.A. Smith, R.B. Wilson, Multi-product pricing for electric power, Energy Econ. 9 (2) (1987) 104–114.

[3] S. Sen, Issues in optimal product design, in: Analytic Approaches to Product and Marketing Planning: The 2nd Conference,1982, pp. 265–274.

[4] S.A. Smith, New product pricing in quality sensitive markets, Mark. Sci. 5 (1) (1986) 70–87.

[5] P. Rusmevichientong, B.V. Roy, P.W. Glynn, A nonparametric approach to multiproduct pricing, Oper. Res. 54 (1) (2006) 82–98.

[6] Aggarwal, T. Feder, R. Motwani, A. Zhu, Algorithms for multi-product pricing, in: Proceedings of the 31st International Colloquium on Automata, Languages and Programming, 2004, pp. 72–83.

[7] V. Guruswami, J.D. Hartline, A.R. Karlin, D. Kempe, C. Kenyon, F. McSherry, On profit-maximizing envy-free pricing, in: Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms, 2005, pp. 1164–1173.

[8] D. Hartline, V. Koltun, Near-optimal pricing in near-linear time, in: Proceedings of the 9th Workshop on Algorithms and Data Structures, 2005, pp. 422–431.

[9] P. Briest, Uniform budgets and the envy-free pricing problem, in: Proceedings of the 35th International Colloquium on Automata, Languages and Programming, 2008, pp. 808–819.

[10] R. Shioda, L. Tunçel, T.G. Myklebust, Maximum utility product pricing models and algorithms based on reservation price, Comput. Optim. Appl. 48 (2) (2011) 157–198.

[11] T. G. J. Myklebust, M. A. Sharpe, L. Tunçel, *Efficient Heuristic Algorithms for Maximum Utility Product Pricing Problems*, Vol. 69 (2012) 25-39.

[12] G. Dobson, S. Kalish, Positioning and pricing a product line, Mark. Sci. 7 (2) (1988) 107–125.

[13] G. Dobson, S. Kalish, Heuristics for pricing and positioning a product-line using conjoint and cost data. Management Science 39 (2) (1993) 160–175.

[14] Cristina G. Fernandes, Carlos E. Ferreira, Álvaro J.P. Franco and Rafael C.S. Schouery, *The envy-free pricing problem, unit-demand markets and connections with the network pricing problem*, Discrete Optimization V. 22 part A (2015) 141-161.

[15] M. Labbé, P. Marcotte, G. Savard, A bilevel model of taxation and its application to optimal highway pricing, Manage. Sci. 44 (12) (1998) 1608–1622.

[16] G. Heilporn, M. Labbé, P. Marcotte, G. Savard, A parallel between two classes of pricing problems in transportation and marketing, J. Revenue Pricing Manag. 9 (1–2) (2009) 110–125.

[17] Library for Efficient Modeling and Optimization in Networks (LEMON) `http://lemon.cs.elte.hu/trac/lemon`

[18] Sedgewick, Robert. Algorithims in C, Third Edition, Part 5, Graph Algorithms.

[19] James C. Bean, Genetic Algorithms and Random Keys for Sequencing and Optimization, ORSA J. Comput. 6 (1994) 154–160.

[20] José Fernando Gonçalves, Mauricio G. C. Resende, Biased random-key genetic algorithms for combinatorial optimization, J. of Heuristics, vol.17, pp. 487-525, 2011.

[21] R. F. Toso, M. G. C. Resende, A C++ application programming interface for biased random-key genetic algorithms, Optimization Methods and Software, vol.30, pp. 81-93, 2015.