RAFAEL LOUBACK FERRAZ

Orientador: Marco Antonio Moreira de Carvalho

EXTENSÃO DA INTERFACE DE PROGRAMAÇÃO DE APLICAÇÕES DO ALGORITMO GENÉTICO DE CHAVES ALEATÓRIAS VICIADAS

Ouro Preto

Dezembro de 2018

Universidade Federal de Ouro Preto
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

EXTENSÃO DA INTERFACE DE PROGRAMAÇÃO DE APLICAÇÕES DO ALGORITMO GENÉTICO DE CHAVES ALEATÓRIAS VICIADAS

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

RAFAEL LOUBACK FERRAZ

Ouro Preto
Dezembro de 2018

F41e Ferraz, Rafael Louback.
Extensão da interface de programação de aplicações do algoritmo genético de chaves aleatórias viciadas [manuscrito] / Rafael Louback Ferraz. - 2018.

79f.: il.: color; grafs; tabs.

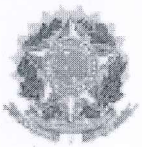
Orientador: Prof. Dr. Marco Antonio Moreira de Carvalho.

Monografia (Graduação). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Departamento de Computação.

1. Algoritmos genéticos. 2. Otimização combinatória. I. Carvalho, Marco Antonio Moreira de. II. Universidade Federal de Ouro Preto. III. Titulo.

CDU: 004.023

Catalogação: ficha.sisbin@ufop.edu.br



UNIVERSIDADE FEDERAL DE OURO PRETO

FOLHA DE APROVAÇÃO

Extensão da Interface de Programação de Aplicações do Algoritmo Genético de Chaves Aleatórias Viciadas

RAFAEL LOUBACK FERRAZ

Monografia defendida e aprovada pela banca examinadora constituída por:

Dr. MARCO ANTONIO MOREIRA DE CARVALHO – Orientador
Universidade Federal de Ouro Preto

Msc. MARCELO LUIZ SILVA
Universidade Federal de Ouro Preto

Msc. REINALDO SILVA FORTES
Universidade Federal de Ouro Preto

Ouro Preto, Dezembro de 2018

Resumo

Na ciência da computação há uma série de problemas de grande interesse prático no dia a dia da sociedade moderna, classificados como problemas de otimização combinatória. Grande parte destes problemas possuem difícil solução em um tempo aceitável. Para alguns, sequer se conhece um método eficiente de solução. Surge então um campo fértil para aplicação de métodos heurísticos e metaheurísticos, que têm se mostrado cada vez mais importantes para a resolução destes problemas. Neste trabalho é estudado o método metaheurístico *Algoritmo Genético de Chaves Aleatórias Viciadas*, um método recente que demonstrou ser eficiente em diversos problemas de complexa solução, principalmente quando o tamanho das instâncias dos problemas aumentam consideravelmente. Este método utiliza técnicas inspiradas na biologia evolutiva e pertence à classe dos algoritmos genéticos, evoluindo soluções de acordo com restrições durante um conjunto finito de gerações, sendo diferenciado principalmente por possuir chaves viciadas e empregar maior chance de perpetuar as características genéticas das melhores soluções ao longo das gerações. Estes conceitos são apresentados em detalhes, bem como uma interface de programação de aplicações relacionada ao método. É proposta a extensão desta interface de programação, pela inclusão de funcionalidades adicionais comuns à utilização do método estudado, levantadas junto à revisão bibliográfica realizada. Espera-se, com a nova interface de programação, contribuir para a reusabilidade de código, agilizando o desenvolvimento de métodos baseados no Algoritmo Genético de Chaves Aleatórias Viciadas, bem como contribuir para o aumento da popularidade do método.

Abstract

In computer science there is a series of problems that modern society has great practical interest to solve on a daily basis, classified as combinatorial optimization problems. Great part of these problems have difficult solution in an acceptable period of time. For some of them, is not known if there is, or could exist, some efficient solution method. Because of that, emerges an entire new field for heuristics and metaheuristics, which had shown to be very efficient at finding good solutions for some of these problems. In this work we study the *Biased Random-Key Genetic Algorithm* (BRKGA) metaheuristic, a recent method that has demonstrated to be efficient in a variety of difficult solution problems, mainly when the size for the instances increases. This method make use of techniques inspired by the evolutionary biology, and is classified as a genetic algorithm, evolving solutions following some problem restrictions until a finite number of generations is developed, being differentiated mainly for the use of biased random keys and providing a greater chance of an offspring solution to inherit genetical characteristics from the best solutions along the generations. These concepts are presented in details, as well the application programming interface related to the method. We propose an extension to that application programming interface, by extending its functionalities based on common extensions found on the literature. With the new programming interface, it is expected to contribute to code reusability, speeding up the development of methods based on the Genetic Algorithm of Biased Random Keys, as well as contributing to the increasing popularity of the method.

Dedico este trabalho a Deus, meus pais Cilas e Leila, minha irmã Ester e todos meus amigos e familiares. Pessoas que são de suma importância em minha vida.

Agradecimentos

Quero agradecer ao meu orientador Marco e a todo corpo docente da UFOP, que me instruiu durante esses anos, no meu desenvolvimento acadêmico.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de snippets de código

Lista de Acrônimos

API *Application Programming Interface*

GRASP *Greedy Randomized Adaptive Search Procedure*

RKGA *Random-Key Genetic Algorithm*

BRKGA *Biased Random-Key Genetic Algorithm*

STL *Standard Template Library*

Capítulo 1

Introdução

Na sua definição original, métodos *Metaheurísticos* são métodos que coordenam métodos para a busca de soluções ótimas locais e estratégias de alto nível para fugir desses ótimos locais. Dessa maneira, é maior a chance de se encontrar uma solução ótima global ou mais próxima dela, sem haver a necessidade de explorar todo o espaço de soluções de um problema (Gendreau e Potvin, 2010). Alguns exemplos de metaheurísticas mais utilizadas são *Greedy Randomized Adaptive Search Procedure* (GRASP), Busca Tabu, Busca Local Iterada e Busca de Vizinhança Variável.

Algoritmos Evolutivos são uma categoria de metaheurísticas inspiradas no conceito da Teoria da Seleção Natural proposta por Darwin (1859). Nesses algoritmos, existe uma série de indivíduos pertencentes a uma população que, devido a uma pressão causada pelo ambiente, somente os mais adaptados participarão das próximas gerações desta população, gerando uma melhor adaptação da população em geral ao ambiente específico (Eiben e Smith, 2003). Esses algoritmos, geralmente, são organizados tal que cada indivíduo da população é uma possível solução válida, e a pressão que o ambiente causaria para estimular a escolha dos mais aptos, usualmente, é uma função objetivo que mede a qualidade dos indivíduos, indicando se este é mais apto ou não para maximizar o valor da solução de um problema. Comumente, também são utilizados mecanismos para criar diversidade nas características dos indivíduos, como, por exemplo, mutação das características de um indivíduo e seleção de alguns indivíduos que não são os melhores de uma geração para reprodução. Com isso, facilita-se escapar de soluções ótimas locais e gera-se maior chance de se aproximar de soluções ótimas globais.

Dentre os algoritmos evolutivos, alguns dos mais utilizados são os *Algoritmos Genéticos*. O método foi proposto originalmente por Holland (1975), e teve como diferencial ser uma técnica que pretendia ser mais genérica ao invés de focar em resolver somente um problema específico, (Mitchell, 1998). O método original proposto consiste em estabelecer uma população inicial de "cromossomos" ou indivíduos (e.g., cadeias de valores binários), tal que cada cromossomo é composto por "genes" (e.g., 0 ou 1). A cada geração, um grupo dos cromossomos é escolhido para se reproduzir e frequentemente os mais aptos (os de melhor valor da função objetivo)

produzem mais descendentes do que os demais. A recombinação (*crossover*) dos genes desses cromossomos é feita imitando o conceito biológico, copiando subpartes (genes) de cada um dos dois cromossomos envolvidos na geração de um descendente. Operadores de mutação também é utilizada para gerar diversidade dentre as novas gerações, aleatoriamente mudando valores dos genes pertencentes a um cromossomo.

Uma variante dos algoritmos genéticos é o *Random-Key Genetic Algorithm* (RKGA) ou Algoritmo Genético de Chaves Aleatórias, proposto por Bean (1994). Seu maior diferencial é utilizar um sistema de chaves aleatórias para codificar as soluções (indivíduos ou cromossomos), em que cada cromossomo é composto por uma sequência de chaves aleatórias, tipicamente no intervalo $[0, 1)$. Essas soluções com chaves aleatórias podem ser decodificadas para o espaço de soluções do problema específico a ser resolvido, possibilitando que o RKGA opere diretamente sobre as chaves e não soluções específicas do problema, criando maior desacoplamento do algoritmo em relação ao problema específico a ser resolvido. Este método também evita o problema de geração de descendentes inviáveis, já que ele representa codificações dos cromossomos de um modo indireto. Essas codificações são avaliadas para sempre gerar novos cromossomos que satisfazem todas as restrições do problema.

Além do uso de chaves aleatórias, o RKGA também utiliza o conceito de *elitismo* para a seleção de quais cromossomos são os "melhores" de uma geração. Por exemplo, são separados em torno de 20% dos cromossomos da atual geração, para serem copiados para a próxima geração, assegurando que as melhores soluções sejam levadas adiante nas novas gerações. Outros 79% são gerados escolhendo-se pares de indivíduos aleatoriamente na geração atual recombinando os seus genes para produzir as novas gerações. A recombinação utiliza *Parametrized Uniform Crossover* (Spears e Jong, 1991), tal que para cada gene é escolhido aleatoriamente de cada cromossomo pai para ser passado ao novo indivíduo. O 1% restante é gerado adicionando-se cromossomos criados de modo aleatório, *mutantes*, à nova geração, criando diversidade e evitando convergência prematura dos indivíduos.

Inspirado no RKGA, foi proposto também o *Biased Random-Key Genetic Algorithm* (BRKGA) ou Algoritmo Genético de Chaves Aleatórias Viciadas. Proposto por Gonçalves e Resende (2010), ele tem operação semelhante ao predecessor, porém ambos diferem em como os pares são escolhidos para reprodução e como a recombinação é aplicada. No BRKGA, a escolha dos cromossomos para a recombinação é aleatória como no RKGA, entretanto, exige-se que um dos escolhidos seja pertencente à elite das soluções, já o outro cromossomo escolhido pode ser de qualquer parte da população. Outro detalhe é que na fase de recombinação do par de indivíduos, o gene provindo do indivíduo da elite deve ter sempre maior chance de ser escolhido sobre o gene do segundo indivíduo.

Para facilitar a utilização do BRKGA, foi desenvolvida por Toso e Resende (2014), uma *Application Programming Interface* (API), ou biblioteca, para a linguagem C++, que implementa a maior parte das rotinas necessárias para a utilização do BRKGA. A cargo do

utilizador da API resta somente implementar a parte específica do problema tratado e ajustar os parâmetros para que o método seja executado adequadamente.

Neste trabalho serão revisadas as extensões comumente utilizadas na literatura sobre o BRKGA, as quais serão implementadas e adicionadas a API existente. Desta forma, uma nova versão da API é proposta neste trabalho, estendida com novas funcionalidades.

1.1 Justificativa

O BRKGA é uma metaheurística recente. Comparações feitas por Gonçalves et al. (2012) demonstraram que, no geral, o BRKGA converge mais rapidamente em direção à solução ótima desejada do que o RKGA. Evidenciando ser uma metaheurística mais eficaz, o BRKGA pode ser empregado em problemas que precisem de soluções rápidas ou em tempo real. As aplicações de sucesso do BRKGA são reportadas em diversas áreas, incluindo Telecomunicações, Transportes, Leilões combinatórios, Problemas de empacotamento, Escalonamento de tarefas e Engenharia de produção em geral.

Toso e Resende (2014) disponibilizaram uma API que inclui a maioria das funcionalidades básicas para implementar o BRKGA com finalidade de resolver um problema de otimização. Entretanto, com o decorrer dos anos, várias funcionalidades extras foram desenvolvidas por diversos autores que podem ser utilizadas com o BRKGA mas que não são implementadas por essa API. Como estas funcionalidades demonstraram ser eficazes, este trabalho foca em implementar essas funcionalidades extras, de forma a auxiliar os usuários desta API, com o intuito de agilizar o desenvolvimento de aplicações que tratem problemas de otimização utilizando o BRKGA, e desejem utilizar tais extensões da metaheuristica.

1.2 Objetivos

De uma maneira geral, este trabalho consiste em realizar pesquisa para geração de embasamento teórico para compreensão dos conceitos relacionados ao tema tratado, bem como o funcionamento da API *brkgaAPI* e suas possíveis extensões de funcionalidades, além de implementar essas novas funcionalidades como extensões da API existente. Além dos objetivos principais, tem-se os seguintes objetivos específicos:

- Compreender os conceitos relacionados a algoritmos genéticos, algoritmos genéticos de chaves aleatórias e algoritmos genéticos de chaves aleatórias viciadas;
- Estudar a organização e a implementação da *brkgaAPI*;
- Pesquisar funcionalidades comuns às aplicações do BRKGA na literatura;
- Implementar de maneira genérica as funcionalidades escolhidas, estendendo a *brkgaAPI*;

- Refinar as implementações realizadas;
- Realizar testes das implementações desenvolvidas;
- Disponibilizar publicamente a versão estendida da *brkgaAPI*.

1.3 Organização do Trabalho

O restante deste texto encontra-se organizado da seguinte forma. O Capítulo 2 apresenta a revisão bibliográfica acerca das aplicações do BRKGA. Em seguida, no Capítulo 3 encontram-se conceitos introdutórios sobre otimização combinatória e a fundamentação teórica sobre os Algoritmos Genéticos e suas variações, com foco no BRKGA. O Capítulo 4 descreve a implementação da *brkgaAPI* original. Em seguida, o Capítulo 5 apresenta a extensão da *brkgaAPI*, com as funcionalidades adicionais propostas neste trabalho. Por fim, as considerações finais deste trabalho são expostas no Capítulo 6.

Capítulo 2

Revisão da Literatura

A metaheurística BRKGA tem sido aplicada a inúmeros problemas de otimização. Neste capítulo, os trabalhos publicados anteriormente são descritos brevemente agrupados por área de aplicação. Para maiores informações sobre uma vasta gama de aplicações do método, pode-se consultar o exame da literatura de autoria de Prasetyo et al. (2015). Extensões do conceitos original do BRKGA podem ser encontradas no trabalho de Lucena et al. (2014).

2.1 Telecomunicações

Os autores Goulart et al. (2011a) utilizaram o BRKGA para minimizar os custos de instalação de uma rede de fibra óptica, descobrindo a configuração das rotas que minimizem a quantidade de dispositivos ópticos utilizados na rede.

Noronha et al. (2011) propuseram o uso do BRKGA para resolver o Problema de Roteamento e Atribuição de Comprimento de Onda para Redes de Fibra Óptica. Propôs-se uma mescla do BRKGA juntamente com métodos conhecidos na literatura e obtiveram-se resultados superiores aos que até então eram considerados estado da arte.

Como demonstrado por Reis et al. (2011), o uso do BRKGA juntamente com os protocolos *Open Shortest Path First* e *Distributed Exponentially-Weighted Flow Splitting*, para resolver o Problema de Atribuição de Pesos para Roteamento em Redes, tem resultados favoráveis. Entre as conclusões, analisou-se que o *Distributed Exponentially-Weighted Flow Splitting* gera menos congestionamento na rede que o *Open Shortest Path First*, mas acarreta maior atraso no tempo de entrega dos pacotes na rede.

Os autores Ruiz et al. (2011) comparam os métodos de Roteamento de Redes *Internet Protocol/Multi-Protocol Label Switching* e *Wavelength Switched Optical Network* de modo a minimizar o investimento em bens de capital. Um algoritmo, baseado no BRKGA, também é proposto visando minimizar os custos e os dois métodos são extensivamente testados.

Pedrola et al. (2013b) confrontam os métodos *Internet Protocol/Multi-Protocol Label Switching* e *Wavelength Switched Optical Network* em uma rede multicamada, visando maximizar a

disponibilidade de serviços na rede enquanto mantém o menor custo possível. Primeiramente, propô-se um método que utiliza programação linear inteira. Em seguida, foi proposto um método GRASP com *Path Relinking* e também outra implementação que utiliza o BRKGA para ajudar a resolver o problema. Experimentos numéricos realizados pelos autores demonstraram que em redes altamente complexas o método GRASP tem eficiência superior ao BRKGA. Além de averiguar que o uso do GRASP com *Path Relinking* melhora consideravelmente a eficiência do algoritmo comparado com o GRASP original.

Posteriormente, Pedrola et al. (2013a) novamente analisaram o uso do GRASP e BRKGA, com variações híbridas envolvendo técnicas de *Path Relinking* e *Variable Neighborhood Descent* para intensificar os resultados anteriores. Os objetivos novamente visam minimizar os custos de implantação e de operação de retransmissores elétricos em redes de fibra óptica. Os resultados demonstram que o uso do BRKGA juntamente com as técnicas de intensificação é viável e mais eficiente do que outras técnicas já conhecidas na literatura.

Goulart et al. (2011b) desenvolveram um método baseado no BRKGA que busca minimizar os custos dos equipamentos necessários para operar uma rede de fibra óptica, determinando o melhor roteamento para essa rede. O método consiste em utilizar o algoritmo estado da arte conhecido na literatura em conjunto com o BRKGA. Os resultados computacionais demonstraram que este novo método gera resultados melhores que o uso do algoritmo que até então era conhecido como estado da arte.

Morán-Mirabal et al. (2013c) contrastam o uso de metaheurísticas, como GRASP com *Path Relinking*, GRASP com *Evolutionary Path Relinking* e o BRKGA para minimizar o *Handover Problem* em redes móveis. Este problema consiste em minimizar as quedas de conexão que um dispositivo móvel possa sofrer quando for transferido de uma torre de transmissão para outra (*handover*). Os autores demonstraram que o método estado da arte encontrado na literatura, até então, só tinha capacidade de resolver instâncias pequenas do problema. Por isso, compararam estes três métodos e notaram que todos eles conseguiram encontrar soluções ótimas para instâncias maiores do problema, mas que o GRASP com *Evolutionary Path Relinking* se demonstrou superior aos demais na maioria dos casos.

Duarte et al. (2014) propõem implementações mais eficientes de heurísticas estabelecidas e o uso dessas heurísticas em conjunto com GRASP e BRKGA, para tratar o problema de minimizar os custos de implantação de retransmissores em redes de fibra óptica. Os resultados demonstraram que o uso do GRASP *Multi-Start* frequentemente encontra soluções de qualidade superior ao uso do GRASP *Single-Start*. GRASP demonstrou gerar soluções de melhor qualidade que o BRKGA. O BRKGA só gera soluções de melhor qualidade se o tempo de execução do algoritmo for superior ao GRASP (maior número de gerações). Ambos os métodos se mostraram mais eficientes que o estado da arte até então.

Resende (2011) apresenta uma analise da literatura sobre as aplicações do BRKGA em telecomunicações. Primeiramente, são introduzidos os conceitos básicos do método, seguido de

uma descrição de metaheurísticas baseadas no BRKGA para roteamento em redes IP, projeto de redes IP com tolerância a falhas, localização de servidores redundantes na distribuição de conteúdo, localização de retransmissores em redes de fibra óptica, e roteamento e alocação de comprimento de ondas em redes de fibra óptica. O autor afirma que para cada problema tratado o BRKGA demonstrou ser uma metaheurística eficiente, frequentemente encontrando soluções de melhor qualidade que de outros métodos conhecidos. Além de que, em alguns casos, este método gerou soluções de qualidade semelhante, mas em menor tempo.

2.2 Transportes

Para o Problema de Congestionamento de Trânsito em Rodovias com Cobrança de Tarifa em Pedágios, Buriol et al. (2010) e Stefanello et al. (2015) utilizaram métodos inspirados no BRKGA para encontrar soluções para grandes instâncias do problema, tal que sejam definidos o local de cobranças na rodovia e também qual valor deve ser atribuído para cobrança com finalidade de reduzir o congestionamento. Os resultados foram favoráveis, encontrando soluções boas, mas nem sempre ótimas para instâncias grandes do problema que não eram possíveis de serem resolvidas com outros métodos.

No *Network Pricing Problem* ou Problema de definição de Preços de Tarifas para Estradas, um problema semelhante ao citado anteriormente, Stefanello et al. (2013) utilizaram o BRKGA em instâncias grandes e diversificadas do problema. Os resultados obtidos pelo autores foram superiores ao método conhecido como estado da arte, sendo capaz de resolver instâncias grandes que o outro método não é capaz de gerar soluções em tempo hábil. A qualidade das soluções obtidas foram próximas do ótimo global, demonstrando que o algoritmo é promissor.

O BRKGA é utilizado por Andrade et al. (2013), para resolver o problema do *K-Interconnected Multi-Depot Multi-Traveling Salesmen*, problema similar ao do Caixeiro Viajante, mas que trata vários vendedores e inclui múltiplos depósitos que adicionam uma dificuldade extra, pois o número de depósitos não é fixo. Os autores compararam o BRKGA com o algoritmo *Multi-Start Heuristic*, que também utiliza busca local. O BRKGA frequentemente apresentou-se levemente mais eficiente que o outro comparado.

Grasas et al. (2014) trataram o Problema de Coleta de Amostras de Sangue para o Transporte até o Laboratório de Análise utilizando o BRKGA. O método se mostrou eficaz nos casos estudados pelos autores, reduzindo o número de viagens necessárias para as coletas.

2.3 Escalonamento

No Problema de Escalonamento de Projetos com Restrições de Recursos, os autores Gonçalves et al. (2010) utilizaram uma variante do BRKGA que utiliza o conhecido *Forward-Backward*

Improvement para melhorar os resultados obtidos inicialmente pela metaheurística. O algoritmo foi extensivamente testado e os resultados foram no geral superiores ou similares a outros algoritmos conhecidos.

Tangpattanakul et al. (2012) utilizaram o BRKGA no Problema de Otimização Multiobjetivo de Seleção e Escalonamento de Observação por Satélites Ágeis de Observação Terrestre. Eles focaram em maximizar o lucro total da operação dos satélites, mas ao mesmo tempo mantendo a disponibilidade para todos os usuários da rede, não deixando os usuários que geram menor lucro sem disponibilidade do serviço. Tangpattanakul et al. (2015) também trata do mesmo problema, utilizando hibridizações do BRKGA em que são consideradas diferentes formas de decodificação e seleção de indivíduos para compor a população elite.

O BRKGA foi empregado no *Job-shop Scheduling Problem* ou problema do escalonamento de processos por Gonçalves e Resende (2011a). No geral o método proposto se demonstrou eficaz quando submetido a testes extensivos.

Gonçalves e Resende (2014) novamente utilizaram o BRKGA hibridizado com outros métodos, para tratar o *Job-Shop Scheduling Problem*. Os resultados obtidos demonstraram já de inicio que 28% das instâncias testadas aferiram resultados de melhor qualidade que os conhecidos até então na literatura. No geral, quando comparado com a outros métodos já estabelecidos na literatura, esta hibridização do BRKGA demonstrou resultados superiores na grande maiorias das classes de instâncias testadas, e foi estatisticamente equivalente na minoria restante.

Moreira et al. (2012) utilizaram o BRKGA para resolver o *Assembly Line Worker Assignment and Balancing Problem* ou Problema de Alocação e Balanceamento de Trabalhadores para Linha de Montagem. O método demonstrou ser tão eficaz quanto os métodos propostos na literatura.

Os autores Araújo et al. (2013) utilizaram o BRKGA para abordar o *Parallel Assembly Line Worker Assignment and Balancing Problem*, mas levando em conta trabalhadores com deficiências físicas de vários tipos e com isso necessidades especiais de trabalho. Os resultados foram promissores já que abordam uma linha de pesquisa sem muitos trabalhos já estabelecidos, mas o BRKGA gerou resultados melhores que outros métodos comparados pelos autores.

Lalla-Ruiz et al. (2014) aplicaram o BRKGA ao Problema de Alocação de Navios em Berços Portuários. O resultados obtidos pelos autores demonstraram que o BRKGA utiliza menos tempo computacional para resolver as instâncias do que os métodos conhecidos na literatura, principalmente quando o tamanho das instâncias aumentavam consideravelmente. O BRKGA também demonstrou ter bom resultados em um cenários onde a quantidade de navios e berços flutuava durante a operação, demonstrando lidar bem com flexibilizações do problema.

Os autores Gonçalves et al. (2016) fizeram o uso do BRKGA hibridizado com busca local

para o Problema de Minimização de Pilhas Abertas. Foram obtidos resultados que corroboram o uso do BRKGA para esse tipo de problema, obtendo resultados equivalentes ou superiores nas instâncias testadas da literatura, quando comparado a outros métodos estabelecidos na literatura.

2.4 Problemas de Agrupamento

O BRKGA foi utilizado pelos autores Roque et al. (2011, 2014) para tratar o *Unit Commitment Problem*. Os resultados obtidos foram promissores, sendo no geral melhores que os resultados obtidos por métodos já estabelecidos como estado da arte. Os resultados foram melhores que os resultados tipicamente obtidos por outros métodos comumente utilizados, e necessitando consideravelmente menos tempo de execução. Também foi observado pouca variância na qualidade das soluções obtidas.

Para o Problema de Agrupamento de Dados, Festa (2013) utilizou o BRKGA e o comparou com diversos algoritmos estado da arte na literatura. O resultados demonstraram que o BRKGA gera soluções de boa qualidade e é apropriado para tratar tal tipo de problema.

Andrade et al. (2014) utilizaram o BRKGA para tratar o *Overlapping Correlation Clustering Problem*. Os resultados apresentados pelos autores demonstraram que o BRKGA gera soluções de qualidade similar a outros métodos considerados estado da arte na literatura. Entretanto, em alguns casos, o tempo de execução exigido é maior que o dos demais métodos, o que segundo o autor é irrelevante pois este tipo de problema não necessita de soluções em tempo real.

2.5 Empacotamento

Gonçalves e Resende (2011b) utilizaram o BRKGA com múltiplas populações paralelas para tratar o *Constrained Two-Dimensional Orthogonal Packing Problem* ou Problema de Empacotamento Ortogonal Bidimensional. Os resultados apresentados pelos autores demonstraram que o BRKGA rendeu resultados semelhantes ou superiores na grande maioria das instâncias testadas contra várias heurísticas conhecidas.

Gonçalves e Resende (2012) aplicaram o BRKGA com múltiplas populações ao Problema de Carregamento de Contêineres, usando pacotes heterogêneos e homogêneos. Os resultados validaram que o método proposto gera soluções de alta qualidade em um tempo aceitável.

No Problema de Empacotamento 2D e 3D, Gonçalves e Resende (2013) utilizam um algoritmo baseado no BRKGA para encontrar a melhor posição para cada pacote dentro de uma caixa para melhor aproveitar o espaço dela. Os resultados demonstraram que frequentemente o método proposto demora menos tempo para chegar a uma solução, além de que no geral, as soluções são de melhor qualidade que de outros métodos.

Kirke et al. (2013) utilizaram o BRKGA para solução do *Multi-Drop Container Loading*, adaptando o algoritmo para uma optimização multi-objetivo. Os autores obtiveram resultados satisfatórios na utilização do algoritmo, acreditando que melhor adaptação do BRKGA para problema possa render melhores resultados no futuro.

Os autores Arefi e Rezaei (2015) propuseram um BRKGA modificado, para resolver o Problema de Carregamento de Contêineres, que representa as chaves aleatórias de modo discreto ao invés de contínuo, para evitar chaves com valores iguais. Foram obtidos resultados superiores quando comparado com outros métodos comumente utilizados, inclusive resultados melhores que do BRKGA tradicional.

Para o Problema de Empacotamento 3D, os autores Zheng et al. (2015) utilizaram uma abordagem de múltiplas populações para o BRKGA, focando em uma optimização biobjetivo. Os resultados obtidos demonstraram que várias soluções perto do ótimo foram encontradas com frequência, aferindo que o método proposto é eficaz, mas que devem ser propostas novas restrições para trabalhos futuros, avaliando a eficiência e eficácia em outras situações de empacotamento que necessite de características mais restritas, como balanceamento de peso dos pacotes e agrupamento de pacotes dentro do contêiner.

Os autores Thomas e Chaudhari (2014) utilizaram o BRKGA para tratar o *2D Strip Packing Problem*. Os resultados obtidos foram encorajadores, superando as melhores heurísticas e metaheurísticas conhecidas na literatura, mesmo quando se trata de instâncias grandes para o problema.

Os autores Chan et al. (2013) e Chan et al. (2015) trataram o *Multi-Item Capacitated Lot-Sizing Problem* utilizando o BRKGA. O método demonstrou ser eficiente para resolver tal tipo de problema, gerando soluções de boa qualidade e utilizando menos tempo de execução que outros métodos conhecidos na literatura.

Gonçalves e Resende (2015) aplicaram o BRKGA puro e hibridizado com técnicas de programação linear no *Unequal Area Facility Layout Problem*. Os resultados obtidos demonstraram que o uso do BRKGA tradicional e o modificado gerou soluções de qualidade superiores as soluções conhecidas na literatura, para a maioria das instâncias conhecidas, utilizando pouco tempo de processamento.

2.6 Recobrimento

Resende et al. (2011) utilizaram um método baseado no BRKGA paralelo com múltiplas populações para resolver o *Steiner Triple Covering Problem*, um problema de cobertura em árvores de alta dificuldade computacional. Para as instâncias dos problemas testados, o algoritmo proposto encontrou as soluções ótimas em todos os casos onde se conhecia a solução ótima. Em inúmeras outras, conseguiu melhorar a solução conhecida até então. O paralelismo utilizado no algoritmo demonstrou amplo aumento na velocidade de execução.

Os autores Hokama et al. (2014) utilizaram o BRKGA para o *Two-Stage Stochastic Steiner Tree Problem*, os resultados obtidos demonstraram que o algoritmo é eficaz para uso em instâncias de difícil solução, que geralmente se apresentam em condições reais de uso. Adicionalmente, no geral o método obtém soluções próximas do ótimo global utilizando muito menos tempo de processamento do que métodos exatos da literatura.

Coco et al. (2014) trataram o *Minmax Relative Regret Robust Shortest Path Problem* utilizando o BRKGA. O algoritmo foi comparado com um método de programação linear inteira e demonstrou encontrar soluções de qualidade bem próximas ao encontrado pelo método exato, mas utilizando menos tempo de execução. O BRKGA também foi comparado pelos autores a outra heurística, que gerou soluções de qualidade equiparáveis mas precisando de menos tempo de execução que ambos os outros métodos.

Ruiz et al. (2015) utilizaram o BRKGA para tratar o *Capacitated Minimum Spanning Tree Problem*. Os resultados demonstraram a eficiência do método proposto, obtendo soluções melhores do que as conhecidas até então na literatura para algumas instâncias, e obtendo soluções de boa qualidade para a maioria das demais instâncias.

2.7 Otimização em Redes

Coco et al. (2012) trataram o Problema do Caminho Mais Curto Robusto utilizando o BRKGA. Os resultados demonstraram que o algoritmo proposto consegue resolver problemas com até mil vértices e demonstrou ser eficiente.

Novamente, uma variante do BRKGA com múltiplas populações foi utilizada por Fontes e Gonçalves (2012) para resolver o *Hop-Constrained Minimum Cost Flow Spanning Tree Problem*. Os resultados apresentados pelos autores demonstram que o algoritmo proposto é com frequência capaz de encontrar soluções próximas do ótimo global, além de ser executado em pouco tempo, mesmo que para grandes instâncias do problema.

Para tratar o *Family Traveling Salesperson Problem*, que é uma variante do conhecido Problema do Caixeiro Viajante, em que o caixeiro precisa visitar um subconjunto dos vértices pertencentes ao grafo para cada um processar tipos diferentes de itens, Morán-Mirabal et al. (2013b) utilizaram o BRKGA e o GRASP com *evolutionary path-relinking* para resolver o problema. A análise feita pelos autores demonstrou que o BRKGA tende a exigir menor tempo computacional para instâncias menores, e que o GRASP se demonstrou mais eficiente para instâncias maiores. Ambos os métodos demonstraram ser muito mais eficientes que o método então estado da arte.

Andrade et al. (2015) fizeram uso do BRKGA no *Wireless Backhaul Network Design Problem* que é similar ao *Steiner Tree Problem* e ao *Facility Location Problem*. Os resultados obtidos pelos autores demonstraram que o BRKGA gera soluções de boa qualidade e pouca variação. Foram superados os métodos estado da arte e até métodos distribuídos comercial-

mente. O BRKGA se destacou principalmente quando utilizada em instâncias maiores, um bom indicativo de que a mesma escala bem em aplicações no mundo real.

No Problema de Gerenciamento de Recursos para Computação em Nuvem ou *Cloud Resource Management Problem* o BRKGA foi utilizado por Heilig et al. (2015). Os autores compararam o método implementado por eles a outros diversos métodos conhecidos na literatura, obtendo resultados superiores com o BRKGA, em pouco tempo de execução.

2.8 Ajuste Automático de Parâmetros em Metaheurísticas

Os autores Festa et al. (2010) utilizaram o BRKGA para definir automaticamente os parâmetros utilizados para executar o GRASP com *path relinking* para resolver o Problema de Atribuição Quadrática Generalizada. Em uma primeira fase, o BRKGA é executado para definir quais os parâmetros geram melhores resultados para a execução do GRASP e, em uma segunda fase, o GRASP é realmente executado. Resultados avaliados pelos autores demonstraram que a solução é viável e robusta.

Morán-Mirabal et al. (2013a) também utilizaram o BRKGA para definir automaticamente os parâmetros para o GRASP com *evolutionary path relinking* para resolver três problemas: Cobertura de Vértices, Corte Máximo e Partição de Vértices em Grafos Capacitados. O resultados demonstrados pelos autores mostram que o GRASP com definição automática de parâmetros pelo BRKGA, geralmente, são executados com menos tempo e geram soluções de qualidade superior que o GRASP com definição de parâmetros manuais.

2.9 Leilões Combinatórios

Variações do BRKGA foram utilizadas por de Andrade et al. (2015) para resolver o *Winner Determination Problem* ou Problema de Determinação do Vencedor em Leilões. Os autores propuseram seis variações do BRKGA e um algoritmo relaxado de inicialização da população para o BRKGA baseado em programação linear inteira. Os resultados publicados demonstram que o BRKGA nem sempre encontra a melhor solução, mas algo bem próximo dela. Porém, este método leva muito menos tempo para ser executado que o estado da arte comparado pelos autores, principalmente, quando comparado em instâncias maiores.

2.10 Otimização Global Contínua

Os autores Silva et al. (2012) utilizaram o BRKGA para resolver problemas de otimização global contínua com restrições de caixa. Foi tratado especificamente o Problema de Cinemática Robótica. O algoritmo foi executado múltiplas vezes, tal que, em cada execução, evitava o espaço de solução que foi encontrado como melhor solução nas execuções anteriores. Os

resultados ilustram que o BRKGA é um opção viável para resolver problemas de otimização global.

O BRKGA foi utilizado por Silva et al. (2013a) para resolver problemas de otimização global contínua com restrições não lineares. Foram realizados testes comparando os resultados com uma variação do GRASP que obteve soluções de boa qualidade, demonstrando a viabilidade do BRKGA para tratar este tipo de problema. Posteriormente, Silva et al. (2013b) utilizaram o BRKGA para tratar problemas de otimização contínua com restrições lineares. Os autores não compararam o método com nenhum outro mas demonstraram seus resultados, tidos como promissores.

Silva et al. (2014) também demonstraram o processo de procurar diversas áreas diferentes no espaço de soluções de sistemas não lineares, tratando o problema de determinação de múltiplas raízes para sistemas de equações não lineares com restrições. Os autores demonstraram que o BRKGA é adequado para ser utilizado para resolver este tipo de problema, obtendo bons resultados. Posteriormente, Silva et al. (2015) desenvolveram uma API distinta para as linguagens Python e C++ que agiliza o desenvolvimento de soluções para problemas de otimização global com restrições utilizando o BRKGA. Os autores descrevem como o método funciona e como se pode compilar, instalar e usar sua API.

Capítulo 3

Fundamentação Teórica

Neste capitulo será apresentada a fundamentação teórica necessária para o entendimento dos conceitos básicos relacionados a otimização e os conceitos relacionados aos algoritmos genéticos e o BRKGA.

3.1 Conceitos Introdutórios

Problemas de Otimização consistem basicamente em se determinar a "melhor" solução entre todas as possíveis soluções válidas com o intuito de alcançar um objetivo, como descrito por Papadimitriou e Steiglitz (1982). Uma *solução* para estes problemas é uma configuração de valores para as variáveis de decisão, dentre as existentes no espaço de soluções específico à um determinado problema. As *variáveis de decisão* são as incógnitas a serem determinadas para solução de um problema, por exemplo, podem indicar quantidades, pertinência ou ordenação de elementos. O *espaço de soluções* é o conjunto de todos os valores ou configurações que as variáveis de decisão do problema possam assumir, sendo composto de um número finito (pequeno ou grande) ou infinito de elementos, dependendo do problema.

Fitness é o conceito que representa o quão adaptado um indivíduo está ao ambiente em que está inserido. Em problemas de otimização, este mesmo termo é utilizado para indicar o valor de uma solução, ou outra métrica de qualidade. Por exemplo, pode ser considerado o quanto um valor da função objetivo relativo a uma solução está próximo do seu melhor valor possível. Considera-se este valor da função objetivo o maior valor possível para problemas de maximização e o menor valor possível para problemas de minimização.

A melhor solução de um problema de otimização, ou *solução ótima global*, é a solução que possui o melhor valor (melhor *fitness*), de acordo com uma função objetivo, dentre todas as possíveis no espaço de soluções. A *função objetivo*, por sua vez, é a função matemática que relaciona os valores atribuídos às variáveis de decisão de uma determinada solução indicando a sua qualidade, maior valor no caso de uma função maximização e menor valor no caso de uma função de minimização. Essa função objetivo deve ser otimizada, respeitando as devidas

restrições que o problema possa impor, em outras palavras, as condições que uma solução deve obedecer para ser *viável*.

Em contraposição à solução ótima global, tem-se as *soluções ótimas locais* ou *subótimas*. Em uma região específica do espaço de soluções, pode haver uma solução dentre toda uma vizinhança de soluções tal que não se pode achar uma outra melhor do que esta. Porém, nem sempre esta solução será a melhor existente em todo espaço de soluções e, sim, somente a melhor em sua vizinhança. Uma *vizinhança*, por sua vez, pode ser definida como um conjunto de soluções para um problema, tal qual é possível relacioná-las dada uma função de proximidade e, com isso, encontrar soluções similares e definir regiões onde estas vizinhanças se encontram no espaço de soluções.

Como exemplo, pode-se ilustrar os conceitos definidos anteriormente com uma instância do *Problema da Mochila Binária*, no qual tem-se uma mochila com uma capacidade máxima de peso suportado e um conjunto de itens com um valor e um peso específicos associados. Deseja-se descobrir a melhor seleção destes itens tal que o maior valor total é atingido, respeitando a capacidade máxima da mochila. Na versão binária deste problema, tem-se a característica de que somente uma unidade de cada item pode ser selecionada para ser colocada na mochila e os itens devem ser escolhidos por inteiro, ou seja, deve ser colocado ou não na mochila.

Neste problema, tem-se as quatro seguintes características:

1. Dados do Problema

   $n =$ Número de itens disponíveis a seleção;

   $i =$ índice para os itens disponíveis, $i = 1, 2, ..., n$;

   $v_i =$ Valor de um item i;

   $w_i =$ Peso de um item i;

   $W =$ Peso máximo suportado pela mochila.

2. Variáveis de Decisão

   $$x_i = \begin{cases} 1, & \text{Se o item } i \text{ foi selecionado para ser colocado na mochila;} \\ 0, & \text{Caso contrário.} \end{cases}$$

3. Função Objetivo

   $$\text{Maximize } \sum_{i=1}^{n} x_i v_i$$

   Como descrito por esta função objetivo, busca-se o maior valor somado dos itens colocados na mochila, em que se tem um somatório das variáveis x_i multiplicadas pelas variáveis v_i.

4. Restrições

$$\sum_{i=1}^{n} x_i w_i \leq W$$

A função objetivo do problema deve obedecer às restrições impostas pelo problema, caso contrário, a maximização da função objetivo seria simplesmente colocar todos os itens na mochila. No problema da mochila binária, tem-se como restrição que a soma dos pesos dos itens colocados na mochila não deve ultrapassar o peso máximo suportado por ela.

No exemplo a seguir, uma instância do problema da mochila binária é composta por quatro itens e uma mochila com capacidade máxima de 10 unidades de peso. A Tabela 3.1 apresenta os itens e seus respectivos valores e pesos.

Tabela 3.1: Itens disponíveis para seleção na mochila

Item	Peso	Valor
1	6	30
2	3	14
3	4	16
4	2	9

A Figura 3.1 apresenta um gráfico que demonstra a evolução do valor que a mochila pode ter para cada solução possível do problema. As soluções no gráfico são representadas como um vetor de valores binários, em que a primeira posição no vetor representa o valor de x_1 e assim por diante até x_n, tal que $n = 4$. Neste gráfico, é possível verificar a relação das possíveis soluções e seus respectivos valores para a função objetivo (*fitness*). Existem dois pontos de ótimo local (amarelo) e um ponto de ótimo global (vermelho), e em cada uma destas soluções apresentadas nota-se que as soluções vizinhas possuem valores piores para a função objetivo. A existência de diferentes ótimos locais motiva a utilização de métodos que não se prendam a ótimos locais e possam diversificar sua busca em outras vizinhanças se for necessário.

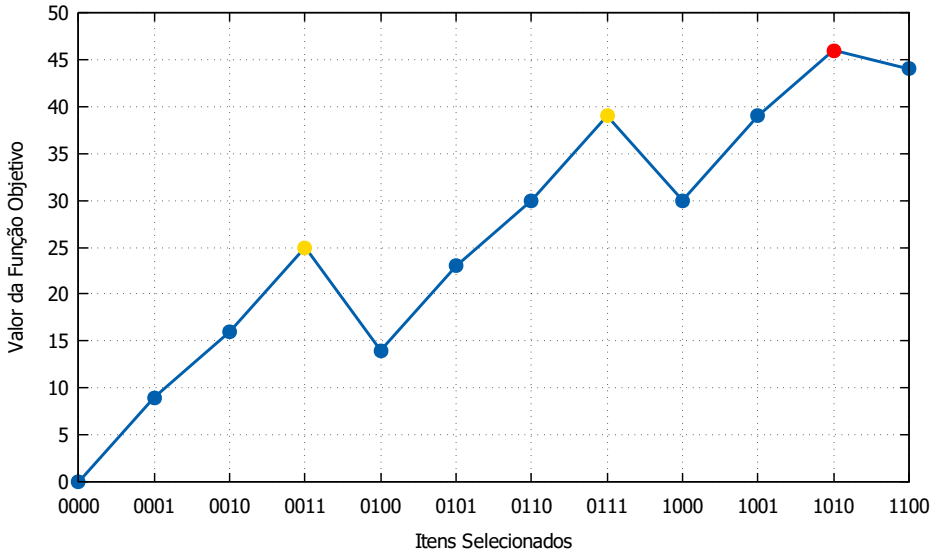


Figura 3.1: Gráfico de soluções da mochila binária: os pontos de ótimo local estão marcados de amarelo e o global de vermelho.

3.2 Algoritmos Genéticos de Chaves Aleatórias Viciadas

Como citado anteriormente, o BRKGA ou algoritmos genéticos de chaves aleatórias viciadas é uma metaheurística evolutiva para problemas de otimização combinatória que se baseia na teoria da evolução de Darwin (1859) para aprimorar uma população de indivíduos até encontrar uma solução ótima ou subótima. Características específicas do BRKGA são a representação de seus indivíduos como vetores de chaves aleatórias e maior probabilidade de passar as características genéticas dos indivíduos pertencentes à elite para as novas gerações.

Este método é composto por cinco etapas. São elas:

1. Geração de chaves aleatórias: inicialização da população, etapa de criação dos vetores de chaves aleatórias;

2. Decodificação de vetores de chaves aleatórias: etapa que codifica os vetores do espaço de chaves aleatórias para o espaço de soluções reais do problema;

3. Classificação da população em elite e não elite: após a geração de uma nova população, esta é ordenada e dividida em dois conjuntos de indivíduos (elite e não elite), de acordo com *fitness* das soluções. A elite é então selecionada para ser copiada para a nova geração sem alterações;

4. Mutação ou geração de mutantes: uma parte da nova geração é formada por indivíduos gerados aleatoriamente. Nesta etapa alguns mutantes são inseridos na nova geração;

5. Cruzamento ou recombinação de indivíduos: o restante dos indivíduos é gerado selecionando aleatoriamente um indivíduo da elite e outro não pertencente à elite, para recombinar os genes dos dois indivíduos, porém, sempre tendendo a escolher um gene do indivíduo pertencente à elite.

Estas etapas são executadas repetidamente até que uma determinada condição de parada seja satisfeita, tal como número máximo de gerações ou convergência da solução para um valor limite. O diagrama apresentado pela Figura 3.2 apresenta o fluxo de execução do BRKGA. Nas seções, a seguir, cada uma destas etapas serão detalhadas e exemplificadas.

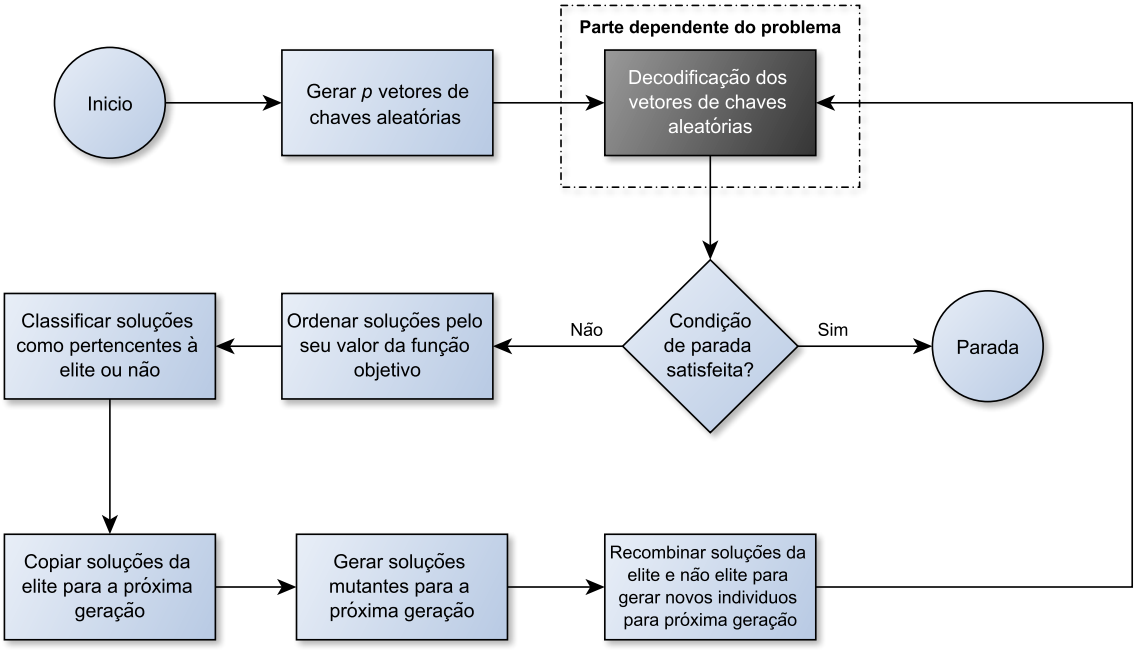


Figura 3.2: Esquema de funcionamento do BRKGA. Adaptado de Gonçalves e Resende (2010).

Para melhor ilustrar o funcionamento do BRKGA será utilizado como exemplo prático o problema da mochila binária descrito na Seção 3.1.

3.2.1 Geração de Chaves Aleatórias

Nesta etapa é inicializada a população a ser evoluída pela metaheurística. Esta população é formada por p vetores de n chaves aleatórias, tal que cada vetor representa um indivíduo ou cromossomo. Cada um dos n genes pertencente ao cromossomo é gerado aleatoriamente por uma chave que pode assumir valores no intervalo de $[0, 1] \in \mathbb{R}$. Na Figura 3.3, pode-se ver um exemplo de um cromossomo representado pelo vetor x.

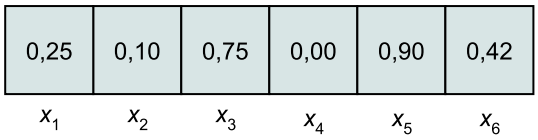


Figura 3.3: Exemplo de cromossomo com seis chaves aleatórias.

A codificação do cromossomo como um vetor de chaves aleatórias, é importante, pois cria desacoplamento da metaheurística em relação ao problema, tal que o algoritmo pode lidar com chaves aleatórias diretamente e somente uma pequena porção do algoritmo trata a parte especifica do problema, aumentando reuso de código e até a utilização de APIs para tal, somente obrigando o usuário da API a implementar esta pequena parte da metaheurística referente à decodificação, vide Figura 3.2.

3.2.2 Decodificação de Indivíduos

Após a inicialização da população, a decodificação dos cromossomos é executada. Esta etapa é a parte com maior dependência do problema a ser tratado, devendo mapear o cromossomo do espaço de solução de chaves aleatórias para o espaço de solução do problema.

Como demonstrado na Figura 3.4, o papel do decodificador é mapear o vetor x (cromossomo) de chaves aleatórias para o espaço de soluções específico ao problema e deve ser implementado de modo distinto em cada problema. Somente depois da fase de decodificação é possível calcular o *fitness* de cada cromossomo.

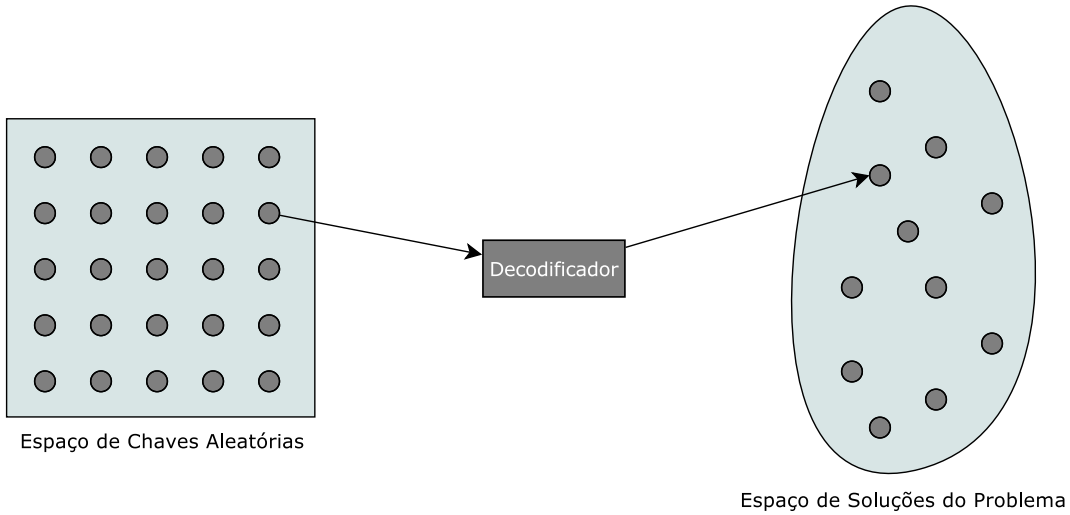


Figura 3.4: Esquema de funcionamento do decodificador, que mapeia as soluções do espaço de chaves aleatórias para o espaço de soluções do problema. Adaptado de Gonçalves e Resende (2010).

Por exemplo, no problema de mochila binária a decodificação pode ser modelada representando os itens selecionados para adição à mochila seguindo a Equação (3.1), de forma que x representa o vetor de chaves aleatórias e x_i a i-ésima chave aleatória a ser decodificada, traduzindo como 1 se o item for selecionado e 0 caso contrário.

$$\begin{cases} x_i < 0,5 \rightarrow 0 \\ x_i \geq 0,5 \rightarrow 1 \end{cases} \tag{3.1}$$

3.2.3 Elitismo

Depois dos cromossomos terem seus respectivos valores de *fitness* calculados na etapa de decodificação, eles são ordenados e posteriormente classificados em uma parcela p_e pertencente à elite que comporta os melhores cromossomos e o restante não pertencente a elite. Essa parcela da população com as melhores soluções é copiada intacta para a próxima geração da população. Com isso, o conceito de elitismo força parcialmente que a metaheurística busque por soluções similares às que estão demonstrando serem as melhores até então. Gonçalves e Resende (2010) afirmam que a proporção recomendada de cromossomos elite na população seja tal que $0,10p \leq p_e \leq 0,25p$ para se obter bons resultados.

Na Figura 3.5, é ilustrada a classificação da população da geração atual, em cromossomos que fazem parte da elite e os que não fazem.

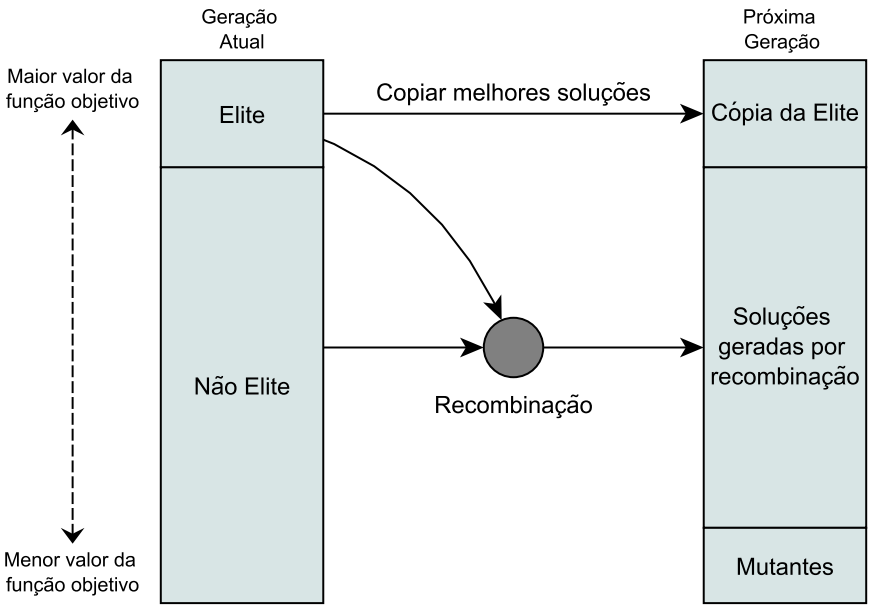


Figura 3.5: Esquema produção de uma nova geração. Adaptado de Gonçalves e Resende (2010).

3.2.4 Mutação

Uma pequena parte da nova geração é produzida aleatoriamente para introduzir diversidade ao processo evolutivo e evitar a convergência prematura para ótimos locais, adicionando cromossomos mutantes na próxima população. Na Figura 3.5, é ilustrada a porção da próxima geração composta por estes cromossomos mutantes. Os autores Gonçalves e Resende (2010) recomendam que a proporção de mutantes na próxima geração seja tal que $0,10p \leq p_m \leq 0,30p$ para que sejam obtidos bons resultados.

3.2.5 Recombinação

Os indivíduos restantes da próxima geração são produzidos na etapa de recombinação. No caso do BRKGA, é utilizado o método *Parametrized Uniform Crossover* (Spears e Jong, 1991), que consiste em sortear para cada gene de qual cromossomos pai deve ser herdado a característica genética. Para que a metaheurística tenha a característica *biased*, ou viciada, um dos cromossomos pai deve ser escolhido aleatoriamente, entretanto, sempre pertencendo à elite e o outro deve ter sua escolha aleatória, porém, advindo do grupo dos que não pertencem a elite. Além disso, no processo de recombinação dos genes, a probabilidade ρ_e de um gene ser herdado do cromossomo da elite deve ser sempre maior ou igual do que a chance de um gene ser herdado do cromossomo não pertencente à elite, ou seja $\rho_e \geq 50\%$.

Na Figura 3.6, é ilustrado o processo de recombinação no BRKGA, tal que dois cromossomos são escolhidos para serem recombinados, seguindo as regras anteriormente citadas e, para cada gene, um número aleatório é sorteado entre 0 e 100. Como tem-se $\rho_e = 70\%$, se o número for menor ou igual a 70, então, é herdado o gene do cromossomo da elite, se for maior que 70, então, o gene é herdado do outro cromossomo.

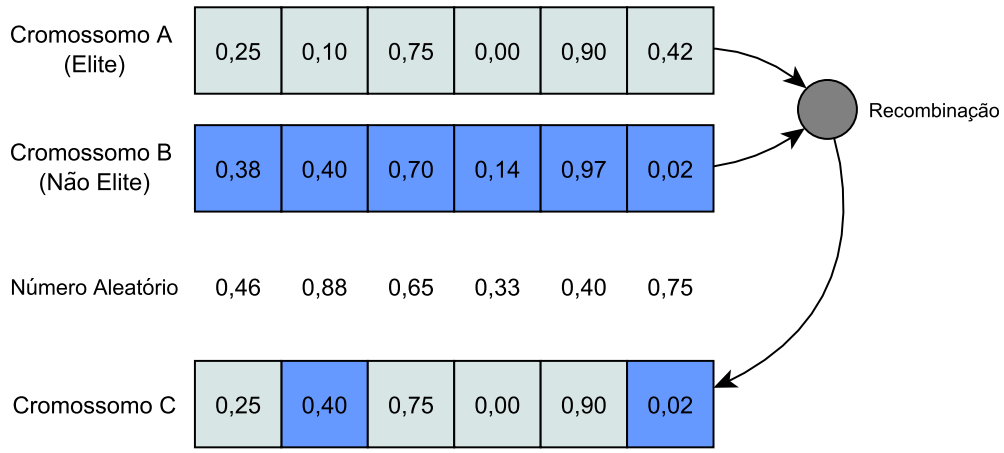


Figura 3.6: Exemplo de recombinação, com $\rho_e = 70\%$. Adaptado de Gonçalves e Resende (2010).

O valores recomendados pelos autores Gonçalves e Resende (2010) para a probabilidade de um gene ser escolhido do progenitor pertencente à elite é de $50\% \leq \rho_e \leq 80\%$.

3.2.6 Condição de Parada

Para que o algoritmo não execute indefinidamente, é necessário determinar qual é a condição que deve ser satisfeita para que a execução termine. Existem vários tipos de condição de parada, dentre elas:

- Número fixo de gerações desde o início da execução do algoritmo;
- Número fixo de gerações desde que houve a última melhora na qualidade da melhor solução;
- Tempo máximo de execução;
- Encontrar uma solução com *fitness* melhor ou igual a um valor estabelecido.

Esta parte do algoritmo é dependente ao problema e deve ser adaptada de acordo as necessidades intrínsecas a ele. Como a escolha da condição de parada geralmente envolve em determinar parâmetros que façam o algoritmo dar a melhor solução dentro de um prazo de tempo ou número máximo de gerações, pode-se utilizar também de paralelização de alguns componentes do algoritmo para acelerar o tempo de execução ou aumentar a chance de se encontrar mais soluções de qualidade.

3.2.7 Múltiplas Populações

O BRKGA pode opcionalmente utilizar o conceito de múltiplas populações. Neste sentido, diferentes populações são evoluídas simultânea e independentemente. Após um determinado número de iterações, essas populações trocam indivíduos de melhor *fitness* entre si (Gonçalves e Resende, 2011b).

Comumente, a troca de cromossomos se dá pela seleção e cópia dos melhores indivíduos de cada população, de modo a substituir os piores indivíduos das demais populações. Esta operação ajuda a melhorar a qualidade geral dos demais indivíduos em cada população, criando maior pressão na área do espaço de busca próximo a esses indivíduos.

3.2.8 Reinício

Luby et al. (1993) introduziram o conceito de reinício do algoritmo de otimização, tal que de acordo com algum critério, a execução pode ser reinicializada. Desta forma, parte-se de um conjunto de dados iniciais, no caso uma nova população gerada aleatoriamente. O reinício pode ser repetido até que um determinado objetivo seja atingido pela execução do algoritmo.

Os autores Luby et al. (1993) definiram que um bom critério para o reinício seria definir um conjunto de instantes de tempo $S = (t_1, t_2, t_3, ...)$, tal que a cada instante dado pela sequência $t_1, t_1 + t_2, t_1 + t_2 + t_3, ...$ marca o reinício da execução do algoritmo. Se os instantes de tempo t_i fossem todos iguais o algoritmo teria uma estrategia de reinício ideal, *i.e.*, $t_1 = t_2 = t_3 = ... = t_i$, em que t_i é uma constante.

3.2.9 Busca Local

Um método comum para implementar melhorias nos indivíduos é a utilização da busca local, como proposto por Buriol et al. (2005), que hibridizaram um algoritmo genético com busca local. Estes métodos podem ser empregados para fazer melhorias a um indivíduo após o processo de cruzamento, aperfeiçoando as caraterísticas de cada indivíduo recém gerado. Estas melhorias são executadas de forma ativa ao invés de utilizar somente os mecanismos próprios dos algoritmos genéticos. Desta maneira, é possível introduzir métodos que considerem as particularidades de cada problema tratado, acelerando a convergência do BRKGA.

Utilizando como exemplo o gráfico da mochila binária dado na Figura 3.1, é possível selecionar uma das possíveis soluções e aplicar um método de busca local para atingir uma solução ótima local. Na Figura 3.7 podemos analisar uma chave aleatória que após ser decodificada gera uma solução específica. Esta por sua vez pode ser utilizada como ponto de partida de uma busca local, a qual eventualmente encontrará uma solução com *fitness* superior ao da solução anterior, melhorando a qualidade dos indivíduos.

3.2.10 Correção de Cromossomos

Após a modificação de algum indivíduo, por exemplo após uma busca local, uma correção deve ser feita para que as chaves aleatórias representem o novo indivíduo modificado. Estas correções são dependentes do problema, sendo submetidas as regras que a decodificação do problema utilize. No gráfico da Figura 3.7 temos a alteração de um valor da solução de 0 para 1. Com isto, é necessária a correção da chave aleatória para que represente este valor, transformando um número menor que $0,5$ em um número maior que este limiar, geralmente, somando $0,5$ ao valor. No caso contrário, subtrai-se $0,5$ do valor original da chave.

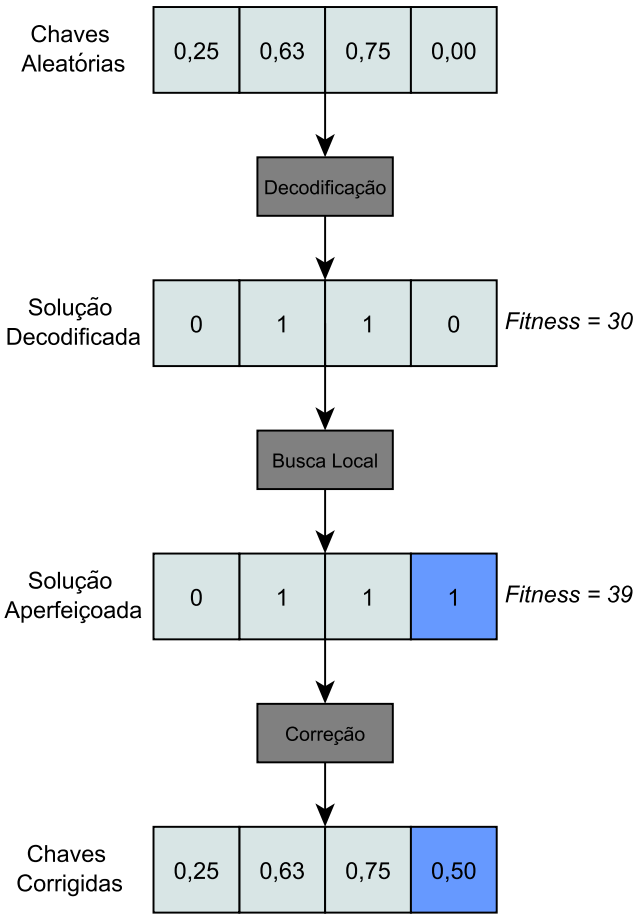


Figura 3.7: Correção de cromossomo utilizando como exemplo o gráfico da mochila binária 3.1.

3.2.11 Aumento da Diversidade na Elite

Durante a execução do BRKGA, é comum que aconteça da porção elite da população convergir rapidamente e se compor por indivíduos muito similares ou até iguais entre si, aumentando a possibilidade que o algoritmo se prenda a ótimos locais. Uma forma de mitigar este problema é avaliar, a cada geração nova produzida, se a população elite está excessivamente similar e alterar o seu conteúdo, em busca de diversificação.

Este processo de diversificação pode se dar de diversas maneiras. Entretanto, uma possível medida consiste em simplesmente excluir uma parte da população pertencente a elite e gerar novos indivíduos mutantes para substituí-los. Este método “empurra” os indivíduos que não pertencem a elite, mas estão próximos dela, a se tornarem indivíduos elite - e ao mesmo tempo - forçando maior diversificação na porção elite e na própria população no geral com a inclusão de novos indivíduos mutantes.

3.2.12 Operadores Dinâmicos

Laoufi et al. (2006) propuseram que alguns operadores dinâmicos podem ser empregados em Algoritmos Genéticos para promover soluções de melhor qualidade e reduzir o tempo para a convergência a resultados aceitáveis ou a solução ótima global. No caso do BRKGA podemos empregar como operadores dinâmicos a fração da população que é classificada como elite p_e, a fração da população que é classificada como mutante p_m e a probabilidade de herança das características do cromossomo pertencente a elite ρ_e. Além do mais, com a manipulação dos valores de p_e e p_m por consequência altera-se a fração dos cromossomos que não pertencem a elite ou aos mutantes. A alteração dos valores destes parâmetros durante a execução do algoritmo possibilita alternar dinamicamente o equilíbrio entre intensificação e diversificação.

Estes parâmetros podem ser ajustados dinamicamente aumentando o valor de p_e quando a população estiver muito diversa e espalhada pelo espaço de soluções, intensificando a busca nas áreas mais promissoras do espaço de soluções. Do mesmo modo, o valor de p_m pode ser aumentado se a população como um todo estiver muito semelhante, diminuindo a probabilidade do algoritmo ficar preso a algum ótimo local e trazendo maior diversidade a esta população. O parâmetro ρ_e pode ser ajustado assim como os demais, aumentando seu valor quando for necessário intensificar a busca no espaço de soluções em regiões próximas a onde se encontra a população elite – ou inversamente – ter seu valor reduzido quando for necessário diminuir a intensidade das buscas nestas regiões.

3.2.13 Paralelização

De acordo com Gonçalves e Resende (2010), o BRKGA tem a capacidade de paralelizar alguns elementos para prover *speedup*[1] na sua execução. Os elementos que são possíveis de serem paralelizados são:

- Geração de chaves aleatórias;
- Geração de cromossomos mutantes para a próxima geração;
- Recombinação;
- Decodificação de vetores de chaves aleatórias;

Os três primeiros não geram tanto impacto na velocidade de execução do algoritmo, porém, o quarto item pode se beneficiar imensamente da paralelização, pois a etapa de decodificação é frequentemente a que mais consome processamento durante a execução da metaheurística. Outro tipo de paralelização aceita pelo método estudado é o uso de múltiplas populações, sendo cada uma delas evoluídas em paralelo e periodicamente mesclando seus cromossomos (Gonçalves e Resende, 2010).

[1] Relação que indica melhora no tempo de execução dentre duas tarefas ou métodos

Capítulo 4

Interface de Programação de Algoritmos Genéticos de Chaves Aleatórias Viciadas

Os autores Toso e Resende (2014) desenvolveram uma Interface de Programação, ou API, que facilita o uso do BRKGA para tratar problemas de otimização combinatória, disponibilizando funções e classes que implementam várias funcionalidades e recursos necessários para a execução da metaheurística. A cargo do usuário da API resta somente adaptar tais recursos para tratar um problema de otimização específico. Comumente, o usuário desta interface de programação somente necessitará de implementar a fase de decodificação específica para o problema tratado, pois todas as outras rotinas do algoritmo serão disponibilizadas pela interface. Neste capítulo, é realizada a descrição detalhada da *brkgaAPI*. O conteúdo é baseado na documentação oficial da mesma, introduzida por Toso e Resende (2014).

A *brkgaAPI* é implementada utilizando a linguagem de programação C++ e a *Standard Template Library* (STL) (Stroustrup, 2000), bem como se faz uso do OpenMP (2017) para disponibilizar suporte ao paralelismo durante a execução do algoritmo. Duas classes principais compõem a *brkgaAPI*, `Population` e `BRKGA`: a primeira define e armazena dados sobre uma população de indivíduos, além de disponibilizar acesso fácil aos mesmos; a segunda define o algoritmo genético e seu funcionamento.

4.1 Visão Geral

Na Figura 4.1 é apresentado o diagrama de classes que ilustra como a API é estruturada. A figura expõe duas classes e duas interfaces: Classe `BRKGA`, Classe `Population`, Interface `Decoder` e Interface `RNG`. Nas próximas seções, estes componentes são descritos em detalhes.

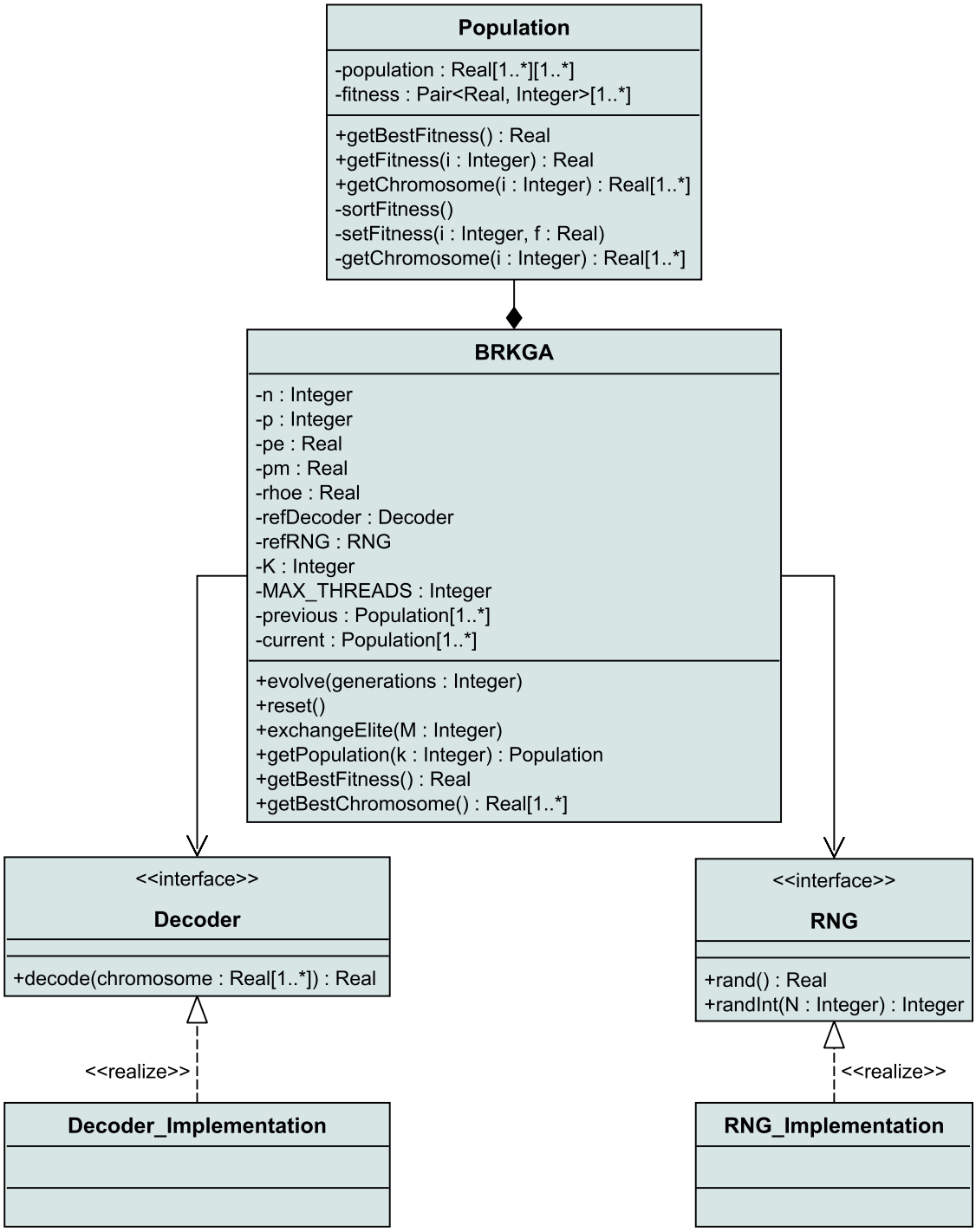


Figura 4.1: Diagrama de Classes da *brkgaAPI*

4.2 Classe `Population`

Esta classe armazena os dados relacionados às representações dos indivíduos e suas características genéticas. Cada indivíduo p é formado por n chaves aleatórias e possui o valor de seu *fitness* determinado pela função objetivo. A implementação do indivíduo é realizada por um objeto arranjo dinâmico da STL, cujo tipo é ponto flutuante de dupla precisão, *i.e*, `std::vector<double>`. A classe `Population` disponibiliza os seguintes métodos para sua manipulação:

- `unsigned getN() const`: retorna o tamanho dos indivíduos em uma população, ou seja, a quantidade de chaves que os compõem;
- `unsigned getP() const`: retorna o tamanho da população;
- `double getBestFitness() const`: retorna o melhor *fitness*, ou seja, o melhor valor de um indivíduo na população;
- `double getFitness(unsigned i) const`: retorna *fitness* do i-ésimo indivíduo. Como a população é mantida ordenada, o parâmetro $i = 0$, obtém o resultado igual a `getBestFitness`;
- `const std::vector<double>& getChromosome(unsigned i) const`: retorna uma referência somente leitura para o i-ésimo indivíduo, ou cromossomo.

4.3 Classe `BRKGA`

Esta classe organiza o funcionamento do algoritmo em si. Por exemplo, é implementado o funcionamento do BRKGA utilizando múltiplas populações independentes, além dos métodos necessários para a manipulação destas populações. Para cada população independente são alocados dois objetos da classe `Population`, um para guardar a população atual e outro para a próxima geração.

A classe BRKGA necessita receber dois componentes externos para funcionar: um decodificador, que transforma a representação de chaves aleatórias em uma solução de um problema específico, criando o desacoplamento do algoritmo em relação ao problema a ser tratado, e um gerador de números aleatórios implementado pelo usuário da *brkgaAPI*, que serve para gerar as chaves aleatórias.

A implementação desta classe é feita por um *template* em C++ definido por:

`template<class Decoder, class RNG> class BRKGA`.

Devido a necessidade destes dois componentes externos, o usuário da *brkgaAPI* deve implementar estas duas interfaces: `Decoder` que implementa as funcionalidades do decodificador, e `RNG` que implementa o gerador de números aleatórios.

4.3.1 Interface `Decoder`

Nesta interface, deve ser implementada exatamente e somente um dos seguintes métodos:

- `double decode(std::vector<double>& chromosome) const`: decodifica um indivíduo ou cromossomo, com acesso de escrita e leitura;
- `double decode(const std::vector<double>& chromosome) const`: decodifica um indivíduo ou cromossomo, com acesso somente de leitura.

Há apenas uma diferença conceitual entre as duas implementações deste método. A primeira implementação permite alterações no indivíduo, dentro da fase de decodificação, e a segunda não as permite.

4.3.2 Interface `RNG`

O componente gerador de números aleatórios necessita que os seguintes métodos sejam implementados:

- `double rand()`: deve retornar um número aleatório de ponto flutuante de dupla precisão variando no intervalo $[0, 1)$;
- `unsigned long randInt(unsigned N)`: deve retornar um número inteiro variando no intervalo $[0, N]$.

A *brkgaAPI* distribui, juntamente com seu código original, a implementação de geração de números aleatórios *Mersenne Twister*, proposta por Matsumoto e Nishimura (1998), que pode ser utilizada opcionalmente em conjunto com a biblioteca.

4.3.3 Manipulação do Algoritmo

Para a operação do BRKGA são disponibilizados na *brkgaAPI* os seguintes métodos:

- `BRKGA(unsigned n, unsigned p, double pe, double pm, double rhoe, const Decoder& refDecoder, RNG& refRNG, unsigned K = 1, unsigned MAX_THREADS = 1)`: construtor da classe BRKGA. A seguir apresenta-se a descrição de cada parâmetro enviado a este construtor:
  - `n`: número de chaves aleatórias em cada indivíduo, ou cromossomo;
  - `p`: número de indivíduos em cada população;
  - `pe`: fração da população que é classificada como pertencente a elite, variando no intervalo $(0, 1)$;

  - `pm`: fração da população que é composta por indivíduos mutantes, adicionados em cada nova geração, variando no intervalo $(0, 1)$;
  - `rhoe`: probabilidade de um indivíduo, gerado pela recombinação de dois outros indivíduos, receber a herança genética do indivíduo pai pertencente à elite. Este valor varia normalmente no intervalo $(0.5, 1)$. Na documentação oficial, este parâmetro é referido como `rho`;
  - `refDecoder`: referência constante para a classe que implementa a interface `Decoder`;
  - `refRNG`: referência para a classe que implementa a interface `RNG`;
  - `K` (opcional, padrão é 1): número de populações independentes;
  - `MAX_THREADS` (opcional, padrão é 1): número máximo de *threads* para realização da decodificação em paralelo. Na documentação oficial, este parâmetro é referido como `MAXT`.

- `evolve(unsigned generations = 1)`: método que evolui as populações seguindo a execução do algoritmo. O parâmetro indica o número de gerações a serem evoluídas;
- `reset()`: reinicia todas as populações com chaves aleatórias novas para cada cromossomo;
- `exchangeElite(unsigned M)`: Realiza M operações de cópia de indivíduos classificados como elite entre diferentes populações, substituindo os M indivíduos com pior *fitness* em cada população;
- `const Population& getPopulation(unsigned k = 0) const`: retorna uma referência constante para a k-ésima população;
- `double getBestFitness() const`: retorna o melhor *fitness* entre todas as populações;
- `const std::vector<double>& getBestChromosome() const`: retorna uma referência constante para o indivíduo, ou cromossomo, com o melhor *fitness* de todas as populações.

Capítulo 5

Extensão da Interface de Programação de Algoritmos Genéticos de Chaves Aleatórias Viciadas

Extensões para a *brkgaAPI*, proposta por Toso e Resende (2014), foram implementadas neste trabalho para dar maior flexibilidade aos usuários desta API. A seleção das funcionalidades foi realizada com base na revisão da literatura apresentada na Seção 2. A seguir, são apresentadas as seis extensões e as modificações que foram efetuadas na API original. Estas extensões contemplam diversificação da população elite, operadores dinâmicos, inicialização heurística da população, hibridização com busca local, correção de cromossomo e configuração automática de parâmetros via o pacote *iRace*.

O código fonte para a implementação das extensões está disponível no repositório `https://github.com/ferrazrafael/BRKGAext`, e está licenciada pela Atribuição-NãoComercial 4.0 Internacional (CC BY-NC 4.0)[1]. Esta licença permite o compartilhamento, adaptação e o uso não comercial, desde que dado o devido crédito de autoria.

5.1 Visão Geral

Na Figura 5.1 é apresentado o diagrama de classes que ilustra como a API estendida foi implementada. A figura expõe três classes e duas interfaces: Classe `BRKGA`, Classe `BRKGAext`, Classe `Population`, Interface `Decoder` e Interface `RNG`. Destaca-se que a classe `BRKGAext` foi adicionada e é derivada da classe `BRKGA`, herdando as suas características para estender as funcionalidades originais implementadas pela *brkgaAPI*.

[1] `https://creativecommons.org/licenses/by-nc/4.0/`

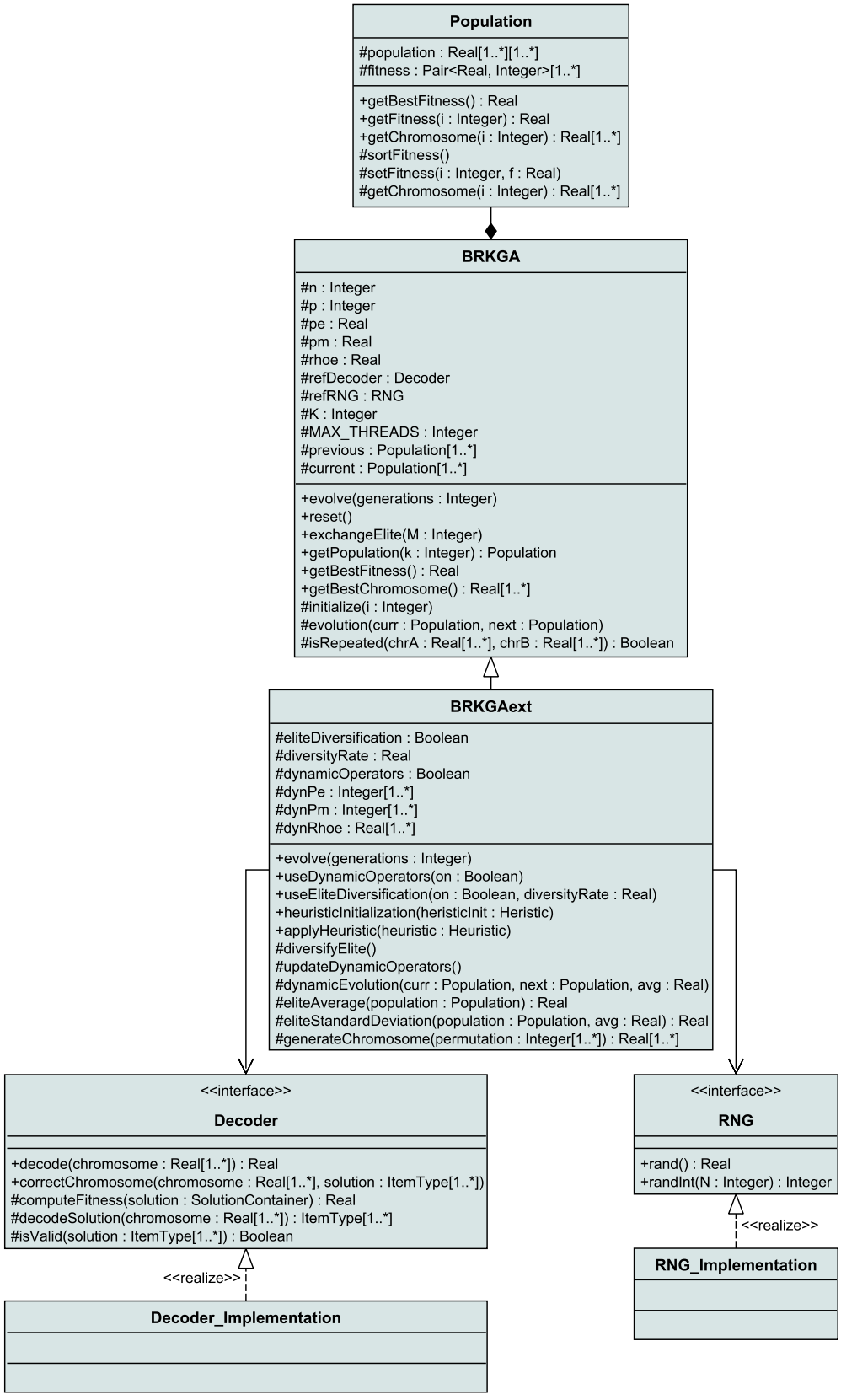


Figura 5.1: Diagrama de Classes da *brkgaAPI* estendida

Na Figura 5.2 é apresentado o fluxo de execução da API estendida, ilustrando como o funcionamento do algoritmo foi alterado com a inclusão das novas funcionalidades. Nas próximas seções, estes componentes são descritos em detalhes.

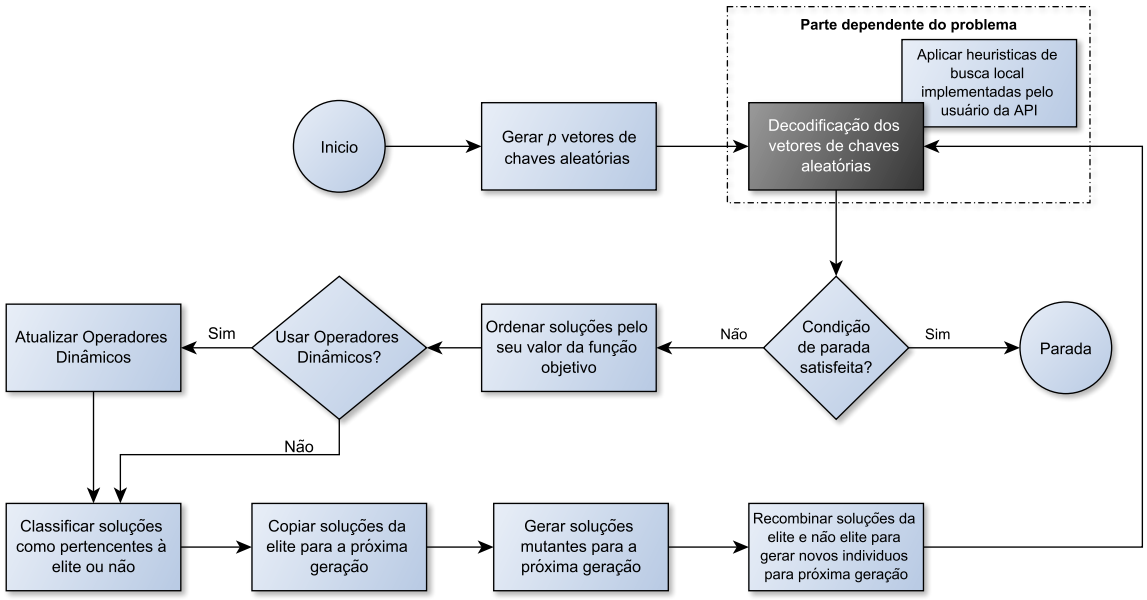


Figura 5.2: Esquema de funcionamento do BRKGA com as extensões implementadas

5.2 Classe `BRKGAext`

Como mencionado, a classe `BRKGAext` é uma classe que herda as características da classe `BRKGA`, descrita no Capítulo 4. Assim como sua classe base, ela fica encarregada de gerir o funcionamento do algoritmo, nesta versão, com as novas funcionalidades. Esta classe é composta pelos seguintes métodos e atributos:

- `void evolve(unsigned generation)`: evolui as populações de geração a geração;
- `void useDynamiOperators(bool on)`: ativa ou desativa o uso de parâmetros adaptativos utilizados no processo de evolução das populações;
- `void useEliteDiversification(bool on, double diversityRate)`: ativa ou desativa o uso de diversificação da elite de cada população;
- `void applyHeuristic(std::function< std::vector<ItemType> (const std::vector<ItemType>&) > heuristic)`: aplica a todas populações uma heurística definida pelo usuário da API, por meio de uma função definida por ele e enviada como parâmetro;

- `void heuristicInitialization(std::function< std::vector<unsigned> (unsigned) > initFunc)`: inicializa as populações utilizando alguma técnica definida pelo usuário da API por meio de uma função definida por ele e enviada como parâmetro;
- `bool eliteDiversification`: indicador para a utilização da diversificação da elite;
- `double diversityRate`: valor da taxa utilizada para diversificação da elite;
- `bool dynamicOperators`: indicador para a utilização de parâmetros adaptativos;
- `std::vector<unsigned> dynPe`: arranjo contendo valor adaptável da fração da população pertencente a elite, para cada população;
- `std::vector<unsigned> dynPm`: arranjo contendo valor adaptável da fração da população pertencente aos mutantes, para cada população;
- `std::vector<double> dynRhoe`: arranjo contendo valor adaptável da probabilidade de um novo indivíduo sendo gerado por cruzamento, receber a herança genética do indivíduo pai pertencente a elite, para cada população;
- `std::vector<double> generateChromosome(const std::vector<unsigned>& permutation)`: gera indivíduo a partir de uma permutação;
- `void diversifyElite()`: aplica diversificação da elite em todas populações;
- `double eliteAverage(const Population& population, unsigned _pe)`: calcula o *fitness* médio da porção elite de uma população;
- `double eliteStandardDeviation(const Population& population, double avg, unsigned _pe)`: calcula o desvio padrão dos *fitness* da porção elite de uma população;
- `void dynamicEvolution(Population& curr, Population& next, unsigned k)`: evolui uma geração de uma população utilizando parâmetros adaptativos;
- `void updateDynamicOperators()`: atualiza os parâmetros adaptativos para todas as populações.

O Snippet de código 5.1 apresenta a declaração da classe `BRKGAext`, conforme descrita. Na sequência, cada um dos métodos é detalhado individualmente.

```
template< class Decoder, class RNG >
class BRKGAext : public BRKGA<Decoder, RNG > {
public:
  BRKGAext(unsigned n, unsigned p, double pe, double pm, double rhoe,
      const Decoder& refDecoder, RNG& refRNG, unsigned K = 1, unsigned MAX_THREADS = 1);
```

```
  virtual ~BRKGAext();

  void evolve(unsigned generations = 1);

  // DynamicOperator
  void useDynamicOperators(bool on);

  // Elite diversification
  void useEliteDiversification(bool on, double diversityRate = 0.05);

  // Apply Heuristic implemented by user
  template <typename ItemType>
  void applyHeuristic(std::function< std::vector<ItemType> (const std::vector<ItemType>&) >
      ↳ heuristic);

  // Initialize population based on initial solution
  void heuristicInitialization(std::function< std::vector<unsigned> (unsigned) >
      ↳ heuristicInit);

protected:
  // Just to shorten base class variable use
  using BRKGA<Decoder, RNG>::n;
  using BRKGA<Decoder, RNG>::p;
  using BRKGA<Decoder, RNG>::pe;
  using BRKGA<Decoder, RNG>::pm;
  using BRKGA<Decoder, RNG>::rhoe;
  using BRKGA<Decoder, RNG>::refRNG;
  using BRKGA<Decoder, RNG>::refDecoder;
  using BRKGA<Decoder, RNG>::K;
  using BRKGA<Decoder, RNG>::MAX_THREADS;
  using BRKGA<Decoder, RNG>::previous;
  using BRKGA<Decoder, RNG>::current;

  // Elite diversification
  bool eliteDiversification = false;
  double diversityRate = 0.0;

  // Adaptive parameters
  bool dynamicOperators = false;
  std::vector<unsigned> dynPe;
  std::vector<unsigned> dynPm;
  std::vector<double> dynRhoe;

  std::vector<double> generateChromosome(const std::vector<unsigned>& permutation);
  void diversifyElite();
  double eliteAverage(const Population& population, unsigned auxPe);
  double eliteStandardDeviation(const Population& population, double avg, unsigned auxPe);
```

```
  void updateDynamicOperators();
  void dynamicEvolution(Population& curr, Population& next, const unsigned k);
};
```

Snippet de código 5.1: Declaração da classe BRKGAext.

5.2.1 Método `evolve`

O método `evolve` reimplementa o método original da classe `BRKGA`, adicionando novas funcionalidades. Inicialmente, a implementação verifica se o uso de operadores dinâmicos está ativado: em caso positivo cada uma das populações é evoluída utilizando o método `dynamicEvolution` e logo em seguida atualiza os operadores dinâmicos; caso contrário invoca-se o método `evolve` original da classe `BRKGA`. Ao final da execução do método é possível invocar o método `diversifyElite` para aplicar a técnica de diversificação da população elite de cada população, se essa funcionalidade estiver ativa. O Snippet de código 5.2 apresenta a implementação deste método.

```
template< class Decoder, class RNG >
void BRKGAext< Decoder, RNG >::evolve(unsigned generations) {
  if(dynamicOperators){
    if(generations == 0) { throw std::range_error("Cannot evolve for 0 generations."); }

    for(unsigned i = 0; i < generations; ++i) {
      for(unsigned j = 0; j < K; ++j) {
        dynamicEvolution(*(current[j]), *(previous[j]), j); // Evolve population
        std::swap(current[j], previous[j]);   // Update generation
      }
    }
    updateDynamicOperators();
  }
  else {
    BRKGA<Decoder, RNG>::evolve(generations);
  }

  if(eliteDiversification) {
    diversifyElite();
  }
}
```

Snippet de código 5.2: Implementação do método BRKGAext::evolve.

5.2.2 Método `applyHeuristic`

O método `applyHeuristic` tem como finalidade aplicar uma heurística definida pelo usuário da API a cada cromossomo de uma população. O método recebe como parâmetro uma função que tenha como assinatura o retorno de um `std::vector<ItemType>` e que receba um parâmetro que também seja um `std::vector<ItemType>`, ou seja, a função enviada como parâmetro deve receber um arranjo de um tipo definido pelo usuário, e retornar outro arranjo com mesmo tipo. O tipo `ItemType` define qual tipo de itens esses arranjos devem conter. O método utiliza esta heurística para tentar melhorar os indivíduos que compõem uma população, aplicando-a a cada uma das soluções decodificadas dos indivíduos. Depois da execução da heurística os cromossomos são corrigidos para fazer com que eles representem as novas soluções encontradas. O Snippet de código 5.3 apresenta a implementação deste método.

```
template<class Decoder, class RNG>
template<typename ItemType>
void BRKGAext<Decoder, RNG>::applyHeuristic(std::function< std::vector<ItemType> (const std::
    ↳ vector<ItemType>&) > heuristic) {
  for(int i = 0; i < int(K); ++i) {
    for(int j = 0; j < int(p); ++j) {
      std::vector<ItemType> solution = refDecoder.decodeSolution((*current[i])(j)) ;
      solution = heuristic(solution);
      refDecoder.correctChromosome((*current[i])(j), solution);
      current[i]->setFitness(j, refDecoder.decode((*current[i])(j)));
    }
  }
}
```

Snippet de código 5.3: Implementação do método BRKGAext::applyHeuristic.

5.2.3 Método `heuristicInitialization`

O método `heuristicInitialization` tem como finalidade inicializar as populações de forma não aleatória. Sendo assim, uma heurística específica para o problema tratado é aplicada sucessivamente, gerando cada um dos indíviduos de uma população. Este método recebe como parâmetro uma função que tenha como assinatura o retorno do tipo `std::vector<unsigned>` e que receba um parâmetro do tipo `unsigned`. Esta função, que será definida pelo usuário da API, deve retornar um arranjo que represente uma permutação de um conjunto de itens, que será utilizada para gerar um cromossomo de modo não aleatório. O Snippet de código 5.4 apresenta a implementação deste método.

```
template< class Decoder, class RNG >
void BRKGAext< Decoder, RNG >::heuristicInitialization(std::function<std::vector<unsigned> (
    ↳ unsigned)> heuristicInit) {
  for(unsigned i = 0; i < K; ++i) {
    for(unsigned j = 0; j < p; ++j) {
      std::vector<unsigned> permutation = heuristicInit(n);
      if(permutation.size() != n) {
        throw std::range_error("Permutation size differs from chromosome size.");
      }

      (*current[i])(j) = generateChromosome(permutation);
    }

    // Decode:
#ifdef _OPENMP
#pragma omp parallel for num_threads(MAX_THREADS)
#endif
    for(int j = 0; j < int(p); ++j) {
      current[i]->setFitness(j, refDecoder.decode((*current[i])(j)) );
    }

    // Sort:
    current[i]->sortFitness();
  }
}
```

Snippet de código 5.4: Implementação do método BRKGAext::heuristicInitialization.

5.2.4 Método `generateChromosome`

O método `generateChromosome` tem como finalidade criar um indivíduo baseado em uma permutação específica e de modo não aleatório. Este é um método auxiliar ao método `heuristicInitialization` descrito anteriormente. O método primeiramente inicializa um arranjo de chaves aleatórias `keys`, para logo em seguida as ordenar. Posteriormente, utiliza-se o arranjo com a permutação que o indivíduo tem que representar para organizar o arranjo final `chromosome` na ordem em que os valores de `keys` devem serem dispostos. Desta forma, quando os genes forem reordenados no futuro, o cromossomo decodificará o indivíduo como a mesma permutação original enviada a este método como parâmetro. O Snippet de código 5.5 apresenta a implementação deste método.

```
template< class Decoder, class RNG >
std::vector<double> BRKGAext< Decoder, RNG >::generateChromosome(const std::vector<unsigned>&
    ↳ permutation) {
  std::vector<double> chromosome(n);
  std::vector<double> keys(n);

  for(unsigned i = 0; i < n; ++i) {
    keys[i] = refRNG.rand();
  }
  std::sort(keys.begin(), keys.end());

  for(unsigned i = 0; i < n; ++i) {
    chromosome[i] = keys[permutation[i]];
  }

  return chromosome;
}
```

Snippet de código 5.5: Implementação do método BRKGAext::generateChromsome.

5.2.5 Método `diversifyElite`

O método `diversifyElite` tem como finalidade diversificar as porções elite das populações. Este método primeiramente calcula o desvio padrão de cada população e qual o percentual esse desvio padrão representa para a população. Posteriormente, se o percentual `percent` for menor do que a taxa de diversidade `diversityRate` enviada pelo usuário como parâmetro, demonstrando que os indivíduos da porção elite são muito similares, será calculado para cada membro pertencente a elite - com exceção do melhor indivíduo - o percentual de desvio de cada indivíduo e se ele for muito parecido com o restante da porção elite, este indivíduo será substituído por um novo mutante. O Snippet de código 5.6 apresenta a implementação deste método.

```
template< class Decoder, class RNG >
void BRKGAext<Decoder, RNG>::diversifyElite() {
  // for each population check if elite is to homogeneous
  for(unsigned i = 0; i < K; ++i) {
    // auxiliary 'pe' that uses dynamic pe if dynamicOperators are on or default pe otherwise
    unsigned _pe = dynamicOperators ? dynPe[i] : pe;

    double avg = eliteAverage(*current[i], _pe);
    double percent = (eliteStandardDeviation(*current[i], avg, _pe) * 100.0) / current[i]->
        ↳ getBestFitness() / 100.0;

    if(percent < diversityRate) {
      // replace elite chromosomes that are below diversity rate with mutants
      for(unsigned j = 1; j < _pe; ++j) {
        double percentualDeviation = fabs( (current[i]->getFitness(j) - avg) / avg );

        if(percentualDeviation < diversityRate) {
          // generating mutant chromosome
          for(unsigned k = 0; k < n; ++k) { (*current[i])(j, k) = refRNG.rand(); }
        }
      }
      // Time to compute fitness, in parallel:
#ifdef _OPENMP
#pragma omp parallel for num_threads(MAX_THREADS)
#endif
      for(int j = 1; j < int(_pe); ++j) {
        current[i]->setFitness( j, refDecoder.decode(current[i]->population[i]) );
      }

      // Now we must sort 'current' by fitness, since things might have changed:
      current[i]->sortFitness();
    }
  }
}
```

Snippet de código 5.6: Implementação do método BRKGAext::diversifyElite.

5.2.6 Método `eliteAverage`

O método `eliteAverage` tem como finalidade calcular a média dos valores de *fitness* dos indivíduos pertencentes a elite de uma população. Este é um método auxiliar ao método `diversifyElite` descrito anteriormente. O Snippet de código 5.7 apresenta a implementação deste método.

```
template< class Decoder, class RNG >
double BRKGAext<Decoder, RNG>::eliteAverage(const Population& population, unsigned _pe) {
  double avg = 0.0;
  for(unsigned i = 0; i < _pe; ++i) {
    avg += population.getFitness(i);
  }
  avg /= double(_pe); // divide sum of elite fitness by auxiliary 'pe'
  return avg;
}
```

Snippet de código 5.7: Implementação do método BRKGAext::eliteAverage.

5.2.7 Método `eliteStandardDeviation`

O método `eliteStandardDeviation` tem como finalidade calcular o desvio padrão dos valores de *fitness* dos indivíduos pertencentes a elite de uma população. Este é um método auxiliar ao método `diversifyElite` descrito anteriormente. O Snippet de código 5.8 apresenta a implementação deste método.

```
template< class Decoder, class RNG >
double BRKGAext<Decoder, RNG>::eliteStandardDeviation(const Population& population, double
    ↳ avg, unsigned _pe) {
  double sum = 0.0;
  for(unsigned i = 0; i < _pe; ++i) {
    sum += population.getFitness(i) - avg;
  }
  sum /= double(_pe); // divide sum of elite fitness by auxiliary 'pe'

  return sqrt(sum);
}
```

Snippet de código 5.8: Implementação do método BRKGAext::eliteStandardDeviation.

5.2.8 Método `dynamicEvolution`

O método `dynamicEvolution` reimplementa o método `evolution`, original da classe `BRKGA`. Este método utiliza os valores dinâmicos de p_e, p_m e ρ_e para fazer o processo de evolução de uma geração de cada uma das populações. A implementação é baseada no método original disponibilizado na *brkgaAPI*, sendo modificadas somente as variáveis previamente relacionadas, que na versão original eram fixas e agora podem ser ajustadas dinamicamente durante a execução do algoritmo. O Snippet de código 5.9 apresenta a implementação deste método.

```
template< class Decoder, class RNG >
void BRKGAext< Decoder, RNG >::dynamicEvolution(Population& curr, Population& next, const
    ↳ unsigned k) {
  // We now will set every chromosome of 'current', iterating with 'i':
  unsigned i = 0; // Iterate chromosome by chromosome
  unsigned j = 0; // Iterate allele by allele

  // 2. The 'pe' best chromosomes are maintained, so we just copy these into 'current':
  while(i < dynPe[k]) {
    for(j = 0 ; j < n; ++j) { next(i,j) = curr(curr.fitness[i].second, j); }

    next.fitness[i].first = curr.fitness[i].first;
    next.fitness[i].second = i;
    ++i;
  }

  // 3. We'll mate 'p - pe - pm' pairs; initially, i = pe, so we need to iterate until i < p
      ↳ - pm:
  while(i < p - dynPm[k]) {
    // Select an elite parent:
    const unsigned eliteParent = (refRNG.randInt(dynPe[k] - 1));

    // Select a non-elite parent:
    const unsigned noneliteParent = dynPe[k] + (refRNG.randInt(p - dynPe[k] - 1));

    // Mate:
    for(j = 0; j < n; ++j) {
      const unsigned sourceParent = ((refRNG.rand() < dynRhoe[k]) ? eliteParent :
          ↳ noneliteParent);

      next(i, j) = curr(curr.fitness[sourceParent].second, j);
    }

    ++i;
  }

  // We'll introduce 'pm' mutants:
  while(i < p) {
    for(j = 0; j < n; ++j) { next(i, j) = refRNG.rand(); }
    ++i;
  }

  // Time to compute fitness, in parallel:
```

Snippet de código 5.9: Implementação do método BRKGAext::dynamicEvolution.

5.2.9 Método `updateDynamicOperators`

O método `updateDynamicOperators` tem como finalidade atualizar os valores de p_e, p_m e ρ_e, para cada uma das populações sendo evoluídas, desde que operadores dinâmicos estejam sendo utilizados. Inicialmente, este método calcula a média dos valores de *fitness* de todos os indivíduos de uma população e altera os valores dos parâmetros que são adaptativos conforme estabelecido por Laoufi et al. (2006). O Snippet de código 5.10 apresenta a implementação deste método.

```
template< class Decoder, class RNG >
void BRKGAext<Decoder, RNG>::updateDynamicOperators() {
  // update operators
  for(unsigned i = 0; i < K; ++i) {
    double avgFitness = 0.0;
    for(unsigned j = 0; j < p; ++j) {
      avgFitness += current[i]->getFitness(j);
    }
    avgFitness /= double(p); // divide sum of all fitness by p

    double bestFitness = current[i]->getBestFitness();

    // generating adaptive pe's
    double k = double(pe) / 2;
    double avg = 0.0;
    for(unsigned j = 0; j < p; ++j) {
      avg += fabs( k * (current[i]->getFitness(j) - bestFitness) / (avgFitness - bestFitness) );
    }
    avg /= double(p); // divide sum of all Pm's by p
    dynPe[i] = unsigned(avg);

    // generating adaptive pm's
    k = double(pm);
    avg = 0.0;
    for(unsigned j = 0; j < p; ++j) {
      avg += fabs( k / (current[i]->getFitness(j) - bestFitness) ) ;
    }
    avg /= double(p); // divide sum of all Pm's by p
    dynPm[i] = unsigned(avg);

    // generating adaptive rhoe's
    avg = 0.0;
    for(unsigned j = 0; j < p; ++j) {
      avg += 0.5 + fabs( 0.25 * (current[i]->getFitness(j) - bestFitness) / (avgFitness - bestFitness) )
          ↳ ;
    }
    dynRhoe[i] = avg / double(p); // divide sum of all Rhoe's by p
  }
}
```

Snippet de código 5.10: Implementação do método BRKGAext::updateDynamicOperators.

5.3 Interface `Decoder`

A interface `Decoder` tem como finalidade separar o código do usuário da API do restante do código genérico do algoritmo, criando desacoplamento do algoritmo e do problema a ser tratado pelo BRKGA. Foi adicionada à *brkgaAPI* a classe `PermutationDecoder` que implementa a interface `Decoder` especificamente para problemas permutacionais. Na Figura 5.3 temos o diagrama da classe `PermutationDecoder`, que descreve sua implementação.

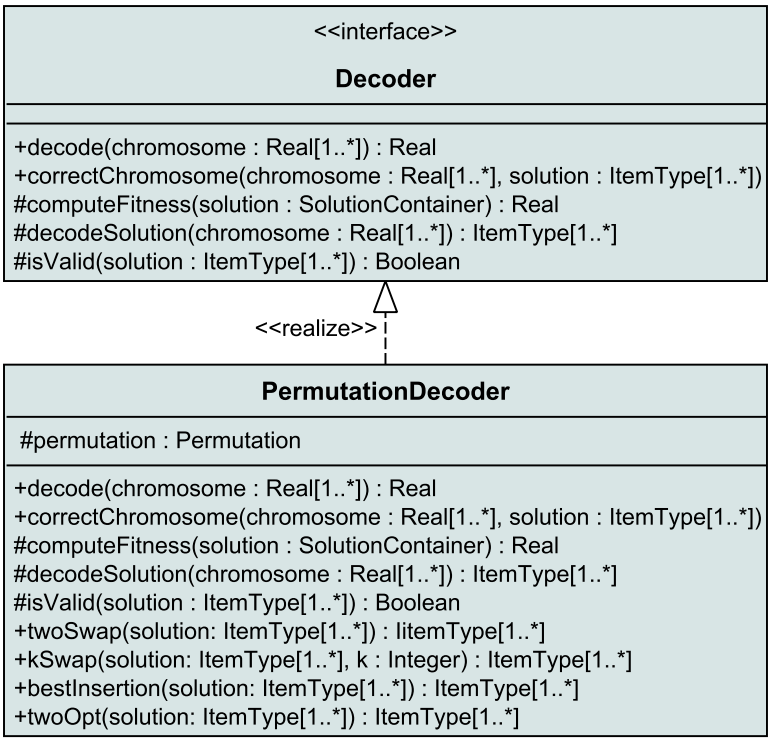


Figura 5.3: Diagrama de Classes da classe `PermutationDecoder`

Estes são os métodos que compõem a classe `PermutationDecoder`:

- `double decode(std::vector<double>& chromosome) const`: decodifica e retorna o *fitness* de indivíduo;
- `std::vector<ItemType> decodeSolution(std::vector<double>& chromosome) const`: decodifica a solução e a retorna, em formato de permutação;
- `std::vector<ItemType> correctChromosome(std::vector<double>& chromosome, const std::vector<ItemType>& solution) const`: corrige a codificação do indivíduo de acordo com uma nova solução válida;

- `std::vector<ItemType> twoSwap(const std::vector<ItemType>& solution)`: implementação da buscal local *2-swap*;
- `const std::vector<ItemType> kSwap(const std::vector<ItemType>& solution, unsigned k)`: implementação da buscal local *k-swap*;
- `std::vector<ItemType> bestInsertion(const std::vector<ItemType>& solution)`: implementação da busca local *Best Insertion*;
- `std::vector<ItemType> twoOpt(const std::vector<ItemType>& solution)`: implementação da buscal local *2-opt*;
- `Permutation permutation`: Classe auxiliar opcional para armazenar os dados do problema;
- `double computeFitness(const SolutionContainer& solution) const`: calcula o *fitness* de uma solução;
- `bool isValid(cosnt std::vector<ItemType>& solution) const`: avalia se indivíduo representa uma solução que não viola alguma restrição;
- `void adjustSolution(std::vector<ItemType>& solution) const`: ajusta uma solução inválida para uma nova solução válida.

O Snippet de código 5.11 apresenta a declaração desta classe, conforme descrita. Na sequência, cada um dos métodos é detalhado individualmente.

```
class PermutationDecoder {
public:
  PermutationDecoder(const Permutation permutation);
  virtual ~PermutationDecoder();

  double decode(std::vector<double>& chromosome) const;
  std::vector<ItemType> decodeSolution(std::vector<double>& chromosome) const;
  void correctChromosome(std::vector<double>& chromosome, const std::vector<ItemType>&
      ↳ solution) const;

  // Local Search
  std::vector<ItemType> twoSwap(const std::vector<ItemType>& solution);
  std::vector<ItemType> kSwap(const std::vector<ItemType>& solution, unsigned k);
  std::vector<ItemType> bestInsertion(const std::vector<ItemType>& solution);
  std::vector<ItemType> twoOpt(const std::vector<ItemType>& solution);

protected:
  const Permutation permutation; // Problem data
```

```
  template <typename SolutionContainer>
  double computeFitness(const SolutionContainer& solution) const;
  bool isValid(const std::vector<ItemType>& solution) const;
  void adjustSolution(std::vector<ItemType>& solution) const;
};
```

Snippet de código 5.11: Declaração da classe PermutationDecoder.

5.3.1 Método `decode`

O método `decode` tem como finalidade decodificar o indivíduo em uma permutação e retornar o seu valor da função objetivo (*fitness*). Uma das modificações efetuadas a interface `Decoder` foi dividir o processo de decodificação em etapas menores, criando outros métodos auxiliares. Estes métodos têm como finalidade possibilitar a execução da correção de cromossomos pela classe `BRKGAext`, além de possibilitar a implementação de métodos de busca local sem a necessidade de total acoplamento do método de busca local com os dados do problema. O Snippet de código 5.12 apresenta a implementação deste método.

```
double PermutationDecoder::decode(std::vector<double>& chromosome) const {
  std::vector<ItemType> solution  = decodeSolution(chromosome);

  return computeFitness(solution);
}
```

Snippet de código 5.12: Declaração do método PermutationDecoder::decode.

5.3.2 Método `decodeSolution`

O método `decodeSolution` tem como finalidade decodificar um indivíduo e retornar a solução que este indivíduo representa. A decodificação é realizada gerando um arranjo de pares em que cada par contenha uma chave do cromossomo e um elemento do problema, e posteriormente, ordenar esse arranjo de pares pelas chaves. Na sequência, copia-se permutação que é gerada pelos elementos reordenados pelos valores das chaves para outro arranjo que represente a solução. Por último o método avalia se a solução é válida, ou seja, se ela não viola nenhuma restrição do problema. Se a solução não for válida, são invocados os métodos `adjustSolution` e `correctChromosome` para ajustar a solução para uma solução válida e depois é efetuada a correção do indivíduo que representa esta nova solução. O Snippet de código 5.13 apresenta a implementação deste método.

```
std::vector<ItemType> PermutationDecoder::decodeSolution(std::vector<double>& chromosome)
    ↳ const {
  std::vector<ItemType> solution = permutation.getItems();
  std::vector< std::pair<double, ItemType> > pairs( permutation.getN() );

  for(unsigned i = 0; i < permutation.getN(); ++i){
    pairs[i].first = chromosome[i];
    pairs[i].second = solution[i];
  }

  // sort by key
  std::sort(pairs.begin(), pairs.end(),
      [](std::pair<double, ItemType> a, std::pair<double, ItemType> b) { return a.first < b.
          ↳ first; } );

  // copy items on order to solution
  std::transform(pairs.begin(), pairs.end(), solution.begin(),
      [](std::pair<double, ItemType> p) { return p.second; } );

   if(!isValid(solution)){
    adjustSolution(solution);
    correctChromosome(chromosome, solution);
  }

  return solution;
}
```

Snippet de código 5.13: Declaração do método PermutationDecoder::decodeSolution.

5.3.3 Método `correctChromosome`

O método `correctChromosome` tem como finalidade de corrigir um indivíduo para representar uma solução que tenha sido decodificada e alterada, por exemplo, por causa da aplicação de uma busca local. A correção é realizada gerando um arranjo de pares em que cada par contém uma chave do cromossomo e um elemento do problema abordado. Posteriormente, esse arranjo de pares é ordenado pelas chaves, e na sequência o método identifica as chaves que mudaram de posição quando a solução foi alterada e as aloca em posições diferentes no cromossomo para representar a nova solução. O Snippet de código 5.14 apresenta a implementação deste método.

```
void PermutationDecoder::correctChromosome(std::vector<double>& chromosome, const std::vector
    ↳ <ItemType>& solution) const {
  std::vector<ItemType> items = permutation.getItems();
  std::vector< std::pair<double, ItemType> > pairs( permutation.getN() );

  for(unsigned i = 0; i < permutation.getN(); ++i){
    pairs[i].first = chromosome[i];
    pairs[i].second = items[i];
  }

  // sort by key
  sort(pairs.begin(), pairs.end(),
      [](std::pair<double, ItemType> a, std::pair<double, ItemType> b) { return a.first < b.
          ↳ first; } );

  for(unsigned i = 0; i < permutation.getN(); ++i)
    if(pairs[i].second != solution[i])
      for(unsigned j = i+1; j < permutation.getN(); ++j)
        if(solution[i] == pairs[j].second)
          std::swap(chromosome[i], chromosome[j]);
}
```

Snippet de código 5.14: Declaração do método PermutationDecoder::correctChromsome.

5.3.4 Método `computeFitness`

O método `computeFitness` tem como finalidade calcular o valor da função objetivo de um indivíduo (*fitness*). Este método está vazio pois deve ser implementado pelo usuário da API, pois o cálculo da função objetivo é dependente do problema, das variáveis e dos dados do problema, não podendo ser implementado de maneira genérica. O uso do `template <typename SolutionContainer>` tem o propósito de dar liberdade ao usuário da API de usar o tipo de estrutura de dados que for mais adequada para atender as suas necessidades, como por exemplo utilizar um `std::vector<ItemType>` para receber uma solução representada por um arranjo, ou também utilizar, por exemplo, um `std::list<ItemType>` para receber uma solução representada por uma lista. O Snippet de código 5.15 apresenta a implementação deste método.

```
template <typename SolutionContainer>
double PermutationDecoder::computeFitness(const SolutionContainer& solution) const {
  // TODO API user implements fitness computation based on problem data
}
```

Snippet de código 5.15: Declaração do método PermutationDecoder::computeFitness.

5.3.5 Método `isValid`

O método `isValid` avalia se uma solução não viola nenhuma restrição do problema sendo tratado, ou seja, se a solução é válida. Como este método depende do problema para ser implementado, ele foi deixado vazio para que o usuário da API o implemente. O Snippet de código 5.16 apresenta a implementação deste método.

```
bool PermutationDecoder::isValid(const std::vector<ItemType>& solution) const {
  // TODO Implement problem requirements validation
}
```

Snippet de código 5.16: Declaração do método PermutationDecoder::isValid.

5.3.6 Método `adjustSolution`

O método `adjustSolution` tem como finalidade ajustar ou corrigir uma solução que não seja válida. Novamente, a implementação deste método depende do problema sendo tratado e das restrições considerados. A implementação foi intencionalmente deixada vazia para que o usuário da API a complete. O Snippet de código 5.17 apresenta a implementação deste método.

```
void PermutationDecoder::adjustSolution(std::vector<ItemType>& solution) const {
  // TODO Implement to adjust invalid solution
}
```

Snippet de código 5.17: Declaração do método PermutationDecoder::adjustSolution.

5.3.7 Método `twoSwap`

O método `twoSwap` implementa a a busca local *2-swap*, que pode ser invocado pelo método `applyHeuristic` da classe `BRKGAext`. Esta clássica busca local de problemas permutacionais consiste em trocar dois elementos de posição sucessivamente, enquanto houver melhoria na função objetivo. O Snippet de código 5.18 apresenta a implementação deste método.

```
std::vector<ItemType> PermutationDecoder::twoSwap(const std::vector<ItemType>& solution) {
  std::vector<ItemType> s = solution;
  std::vector< std::pair<unsigned, unsigned> > pairs(permutation.getN());

  for(unsigned i = 0; i < permutation.getN(); ++i){
    for(unsigned j = i+1; j < permutation.getN(); ++j){
      pairs[i].first = i;
      pairs[i].second = j;
    }
  }

  std::random_shuffle(pairs.begin(), pairs.end());

  double bestFitness = computeFitness(solution);
  for(unsigned i = 0; i < pairs.size();){
    // swap
    unsigned a = pairs[i].first;
    unsigned b = pairs[i].second;
    std::swap(s[a], s[b]);

    if(isValid(s)){
      double fitness = computeFitness(s);
      if(fitness > bestFitness){
        bestFitness = fitness;
        std::random_shuffle(pairs.begin(), pairs.end());
        i = 0;
      }
      else{
        // undo swap
        std::swap(s[b], s[a]);
        ++i;
      }
    }
    else{
      // undo swap
      std::swap(s[b], s[a]);
      ++i;
    }

  }

  return s;
}
```

Snippet de código 5.18: Declaração do método PermutationDecoder::twoSwap.

5.3.8 Método `kSwap`

O método `kSwap` implementa a a busca local *k-swap*, que pode ser invocado pelo método `applyHeuristic` da classe `BRKGAext`. Esta busca local é uma generalização da busca local descrita anteriormente, na qual são trocados k pares de elementos. O Snippet de código 5.19 apresenta a implementação deste método.

```
std::vector<ItemType> PermutationDecoder::kSwap(const std::vector<ItemType>& solution,
    ↳ unsigned k) {
  std::vector<ItemType> s = solution;
  std::vector< std::pair<unsigned, unsigned> > pairs(permutation.getN());
  for(unsigned i = 0; i < permutation.getN(); ++i)
    for(unsigned j = i+1; j < permutation.getN(); ++j)
      { pairs[i].first = i; pairs[i].second = j; }
  random_shuffle(pairs.begin(), pairs.end());
  double bestFitness = computeFitness(solution);
  for(unsigned i = 0; i < pairs.size();){
    for(unsigned j = 0, c = i; j < k; ++j, ++c){
      unsigned x = c % pairs.size(), a = pairs[x].first, b = pairs[x].second;
      std::swap(s[a], s[b]);
    }
    if(isValid(s)){
      double fitness = computeFitness(s);
      if(fitness > bestFitness){
        bestFitness = fitness; std::random_shuffle(pairs.begin(), pairs.end());
        i = 0;
      }
      else{
        for(unsigned j = k+1, c = i+j-1; j > 0; --j, --c){
          unsigned x = c % pairs.size(), a = pairs[x].first, b = pairs[x].second;
          std::swap(s[b], s[a]);
        }
        ++i;
      }
    }
    else{
      for(unsigned j = k+1, c = i+j-1; j > 0; --j, --c){
        unsigned x = c % pairs.size(), a = pairs[x].first, b = pairs[x].second;
        std::swap(s[b], s[a]);
      }
      ++i;
    }
  }
  return s;
}
```

Snippet de código 5.19: Declaração do método PermutationDecoder::kSwap.

5.3.9 Método `bestInsertion`

O método `bestInsertion` implementa a a busca local *Best Insertion*, que pode ser invocado pelo método `applyHeuristic` da classe `BRKGAext`. Esta também consiste e clássica busca local de problemas permutacionais que se baseia em remover um elemento da permutação por vez e o inserir na posição que gerar o melhor valor de acordo com a função objetivo. Todos os elementos da permutação são submetidos a esta operação. O Snippet de código 5.20 apresenta a implementação deste método.

```
std::vector<ItemType> PermutationDecoder::bestInsertion(const std::vector<ItemType>& solution
    ↳ ) {
  std::list<ItemType> sList(solution.begin(), solution.end()); // copies solution elements to
      ↳  s list

  std::vector<ItemType> items = solution;
  std::random_shuffle(items.begin(), items.end());

  std::vector<ItemType> s = solution;
  double bestFitness = computeFitness(solution);
  for(unsigned i = 0; i < items.size(); ++i){
    std::list<ItemType>::iterator pos = find(sList.begin(), sList.end(), items[i]);
    sList.erase(pos);

    for(std::list<ItemType>::iterator it = sList.begin(); it != sList.end(); ++it){
      sList.insert(it, items[i]);
      double fitness = computeFitness(sList);
      if(fitness > bestFitness){
        bestFitness = fitness;
        std::copy(sList.begin(), sList.begin(), s.begin());
      }
      sList.remove(items[i]);
    }

    sList.insert(--pos, items[i]);
  }

  return s;
}
```

Snippet de código 5.20: Declaração do método PermutationDecoder::bestInsertion.

5.3.10 Método `twoOpt`

O método `twoOpt` implementa a a busca local *2-Opt*, que pode ser invocado pelo método `applyHeuristic` da classe `BRKGAext`. Originamente projetada para o problema do caixeiro viajante, esta busca local seleciona aleatoriamente duas posições de uma permutação e inverte as posições dos elementos neste intervalo sucessivamente, enquanto houver melhoria na função objetivo. O Snippet de código 5.21 apresenta a implementação deste método.

```
std::vector<ItemType> PermutationDecoder::twoOpt(const std::vector<ItemType>& solution) {
  std::vector<ItemType> s = solution;
  std::vector< std::pair< std::vector<ItemType>::iterator, std::vector<ItemType>::iterator> >
      ↳ pairs;
  pairs.reserve(s.size());

  for(std::vector<ItemType>::iterator itA = s.begin(); itA != s.end(); ++itA){
    for(std::vector<ItemType>::iterator itB = next(itA); itB != s.end(); ++itB){
      pairs.push_back( make_pair(itA, itB) );
    }
  }

  std::random_shuffle(pairs.begin(), pairs.end());

  double bestFitness = computeFitness(solution);
  for(unsigned i = 0; i < pairs.size(); ++i){
    auto itA = pairs[i].first;
    auto itB = pairs[i].second;
    std::reverse(itA, itB);

    if(isValid(s)){
      double fitness = computeFitness(s);
      if(fitness > bestFitness)
        bestFitness = fitness;
      else
        std::reverse(itB, itA);
    }
    else{
      std::reverse(itB, itA);
    }
  }

  return s;
}
```

Snippet de código 5.21: Declaração do método PermutationDecoder::twoOpt.

5.4 Ajuste Automático de Parâmetros

Conforme mencionado no Capítulo 3, o *BRKGA* possui um conjunto de parâmetros que precisam ser ajustados para execução adequada do método. Uma maneira de ajustar automaticamente os valores dos parâmetros é por meio da ferramenta *iRace* (López-Ibáñez et al., 2016), um pacote codificado na linguagem R. O *iRace* é um método *offline* de configuração automática de algoritmos de otimização que, dado um conjunto de instâncias de um problema específico e um conjunto de possíveis valores para os parâmetros, determina uma combinação apropriada de valores para os parâmetros.

Para que *iRace* interaja com qualquer método computacional, é necessário que o código do método seja adequado para receber os possíveis valores de cada parâmetro como argumento de linha de comando, e deve retornar o resultado em um formato específico. Juntamente com as extensões propostas neste trabalho, incluiu-se uma função específica para ajuste de parâmetros do BRKGA via *iRace*. Também são fornecidos todos os arquivos de configuração do *iRace* para ajuste dos parâmetros tamanho da população, percentual da população elite, percentual da população mutante, probabilidade de herdar genes do pai elite, número máximo de gerações, intervalo de troca de cromossomos e quantidade de cromossomos trocados. No Snippet de código 5.22 é apresentada a implementação desta função, lembrando que esta é uma `main` alternativa que só utilizada para a utilização do *iRace*, devendo ser substituída pela original depois que os parâmetros forem estabelecidos.

```
int main(int argc, char* argv[]) {
  unsigned n, p;
  double pe, pm, rhoe;
  unsigned K, X_INTVL, X_NUMBER, MAX_GENS;
  const unsigned MAXT = 2;  // number of threads for parallel decoding

  vector<string> arguments(argv + 1, argv + argc);
  for(unsigned i=1; i<arguments.size(); i+=2)
  {
    string::size_type sz;
    if(arguments[i]== "--p")
      p = n*stoi(arguments[i+1], &sz);
    else if(arguments[i]== "--pe")
      pe = stod(arguments[i+1], &sz);
    else if(arguments[i]== "--pm")
      pm = stod(arguments[i+1], &sz);
    else if(arguments[i]== "--rhoe")
      rhoe = stod(arguments[i+1], &sz);
    else if(arguments[i]== "--K")
      K = stoi(arguments[i+1], &sz);
    else if(arguments[i]== "--X_INTVL")
      X_INTVL = stoi(arguments[i+1], &sz);
```

```
    else if(arguments[i]== "--MAX_GENS")
      MAX_GENS = stoi(arguments[i+1], &sz);
    else if(arguments[i]== "--X_NUMBER")
      X_NUMBER = stoi(arguments[i+1], &sz);
  }

  const long unsigned rngSeed = 0;  // seed to the random number generator
  MTRand rng(rngSeed);        // initialize the random number generator

  vector<unsigned> items = {1, 2, 3, 4, 5};     // items of the permutation
  Permutation permutation(items, items.size());
  PermutationDecoder decoder(permutation);
  BRKGAext< PermutationDecoder, MTRand > algorithm(n, p, pe, pm, rhoe, decoder, rng, K, MAXT)
      ↳ ; // initialize the decoder

  unsigned generation = 0;    // current generation
  do {
    algorithm.evolve(); // evolve the population for one generation

    if((++generation) % X_INTVL == 0) {
      algorithm.exchangeElite(X_NUMBER);  // exchange top individuals
    }
  } while (generation < MAX_GENS);

  return unsigned( algorithm.getBestFitness() );
}
```

Snippet de código 5.22: Adaptação do `main` para a API ser utilizada com *iRace*

Capítulo 6

Conclusões

Neste trabalho foram introduzidos os conceitos referentes aos algoritmos genéticos e algumas de suas variantes, particularmente o Algoritmo Genético de Chaves Aleatórias Viciadas (ou *Biased Random-Key Genetic Algorithm*, BRKGA). Também foi revisada a literatura referente ao BRKGA, assim como suas aplicações nas mais diversas áreas, além de efetuado um estudo detalhado do funcionamento desta metaheurística. Devido a este levantamento, é possível concluir que o BRKGA demonstra ser eficiente em vários tipos de problemas e se mostra uma ferramenta interessante para a aplicação em outros problemas diversos.

Recentemente foi proposta uma API, denominada *brkgaAPI*, que implementa as funcionalidades básicas para utilização do BRKGA. Neste trabalho foram analisadas e implementadas seis extensões a esta API, incluindo funcionalidades relativas a diversificação da população elite, operadores dinâmicos, inicialização heurística da população, hibridização com busca local, correção de cromossomo e configuração automática de parâmetros via o pacote *iRace*. A seleção das extensões a serem adicionadas foi realizada com base na revisão da literatura. Os componentes propostos foram cuidadosamente implementados e testados, de maneira que a API estendida contribui para a agilidade de codificação e reusabilidade de código ao prover funcionalidades comuns a usuários da API original, sem a necessidade de o usuário envolver-se em detalhes da implementação. A nova versão da *brkgaAPI* está disponível publicamente na internet para livre uso não comercial[1]. Trabalhos futuros incluem experimentos computacionais extensivos que incluam a aplicação da API estendida a algum problema previamente abordado pela aplicação da API original do BRKGA e comparação posterior com a utilização das novas funcionalidades das extensões propostas.

[1] `https://github.com/ferrazrafael/BRKGAext`

Referências Bibliográficas

Andrade, C. E.; Miyazawa, F. K. e Resende, M. G. (2013). Evolutionary algorithm for the k-interconnected multi-depot multi-traveling salesmen problem. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pp. 463–470, New York, NY, USA. ACM.

Andrade, C. E.; Resende, M. G.; Karloff, H. J. e Miyazawa, F. K. (2014). Evolutionary algorithms for overlapping correlation clustering. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, pp. 405–412, New York, NY, USA. ACM.

Andrade, C. E.; Resende, M. G.; Zhang, W.; Sinha, R. K.; Reichmann, K. C.; Doverspike, R. D. e Miyazawa, F. K. (2015). A biased random-key genetic algorithm for wireless backhaul network design. *Appl. Soft Comput.*, 33(C):150–169.

Araújo, F. F. B.; Costa, A. M. e Miralles, C. (2013). Balancing parallel assembly lines with disabled workers. *CoRR*, abs/1304.1423.

Arefi, M. S. e Rezaei, H. (2015). Problem solving of container loading using genetic algorithm based on modified random keys. *Journal of Advanced Computer Science & Technology*, 4(1).

Bean, J. C. (1994). Genetic algorithms and random keys for sequencing and optimization. *INFORMS Journal on Computing*, 6(2):154–160.

Buriol, L.; Resende, M.; Ribeiro, C. e Thorup, M. (2005). A hybrid genetic algorithm for the weight setting problem in ospf/is-is routing. *Networks*, 46:36–56.

Buriol, L. S.; Hirsch, M. J.; Pardalos, P. M.; Querido, T.; Resende, M. G. C. e Ritt, M. (2010). A biased random-key genetic algorithm for road congestion minimization. *Optimization Letters*, 4(4):619–633.

Chan, F. T. S.; Tibrewal, R. K.; Prakash, A. e Tiwari, M. K. (2013). *Inventory Based Multi-Item Lot-Sizing Problem in Uncertain Environment: BRKGA Approach*, pp. 1197–1206. Springer International Publishing, Heidelberg.

Chan, F. T. S.; Tibrewal, R. K.; Prakash, A. e Tiwari, M. K. (2015). A biased random key genetic algorithm approach for inventory-based multi-item lot-sizing problem. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, 229(1):157–171.

Coco, A. A.; Júnior, J. C. A.; Noronha, T. F. e Santos, A. C. (2014). An integer linear programming formulation and heuristics for the minmax relative regret robust shortest path problem. *Journal of Global Optimization*, 60(2):265–287.

Coco, A. A.; Noronha, T. F. e Santos, A. C. (2012). A biased random-key genetic algorithm for the robust shortest path problem. In *Proceedings of Global Optimization Workshop (GO2012)*, pp. 53–56.

Darwin, C. (1859). *On the origin of species*. New York :D. Appleton and Co.,.

de Andrade, C. E.; Toso, R. F.; Resende, M. G. C. e Miyazawa, F. K. (2015). Biased random-key genetic algorithms for the winner determination problem in combinatorial auctions. *Evolutionary computation*, 23(2):279–307.

Duarte, A.; Martí, R.; Resende, M. e Silva, R. (2014). Improved heuristics for the regenerator location problem. *International Transactions in Operational Research*, 21(4):541–558.

Eiben, A. E. e Smith, J. E. (2003). *Introduction to Evolutionary Computing*. SpringerVerlag.

Festa, P. (2013). A biased random-key genetic algorithm for data clustering. *Mathematical Biosciences*, 245(1):76 – 85. SI : BIOCOMP 2012.

Festa, P.; Gonçalves, J. F.; Resende, M. G. C. e Silva, R. M. A. (2010). *Experimental Algorithms: 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings*, chapter Automatic Tuning of GRASP with Path-Relinking Heuristics with a Biased Random-Key Genetic Algorithm, pp. 338–349. Springer Berlin Heidelberg, Berlin, Heidelberg.

Fontes, D. B. M. M. e Gonçalves, J. F. (2012). A multi-population hybrid biased random key genetic algorithm for hop-constrained trees in nonlinear cost flow networks. *Optimization Letters*, 7(6):1303–1324.

Gendreau, M. e Potvin, J.-Y. (2010). *Handbook of Metaheuristics*. Springer Publishing Company, Incorporated, 2ª edição.

Gonçalves, J. F. e Resende, M. G. (2014). An extended akers graphical method with a biased random-key genetic algorithm for job-shop scheduling. *International Transactions in Operational Research*, 21(2):215–246.

Gonçalves, J. F. e Resende, M. G. (2015). A biased random-key genetic algorithm for the unequal area facility layout problem. *European Journal of Operational Research*, 246(1):86 – 107.

Gonçalves, J. F.; Resende, M. G. e Costa, M. D. (2016). A biased random-key genetic algorithm for the minimization of open stacks problem. *International Transactions in Operational Research*, 23(1-2):25–46.

Gonçalves, J. F. e Resende, M. G. C. (2010). Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, 17(5):487–525.

Gonçalves, J. F. e Resende, M. G. C. (2011a). A biased random-key genetic algorithm for job-shop scheduling. *AT&T Labs Research Technical Report*, 46:253–271.

Gonçalves, J. F. e Resende, M. G. C. (2011b). A parallel multi-population genetic algorithm for a constrained two-dimensional orthogonal packing problem. *Journal of Combinatorial Optimization*, 22(2):180–201.

Gonçalves, J. F.; Resende, M. G. C. e Mendes, J. J. M. (2010). A biased random-key genetic algorithm with forward-backward improvement for the resource constrained project scheduling problem. *Journal of Heuristics*, 17(5):467–486.

Gonçalves, J. F. e Resende, M. G. (2012). A parallel multi-population biased random-key genetic algorithm for a container loading problem. *Computers & Operations Research*, 39(2):179 – 190.

Gonçalves, J. F. e Resende, M. G. (2013). A biased random key genetic algorithm for 2d and 3d bin packing problems. *International Journal of Production Economics*, 145(2):500 – 510.

Gonçalves, J. F.; Resende, M. G. C. e Toso, R. F. (2012). Biased and unbiased random-key genetic algorithms: An experimental analysis. Technical report, AT&T Labs Research, Florham Park.

Goulart, N.; de Souza, S.; Dias, L. e Noronha, T. (2011a). Biased random-key genetic algorithm for fiber installation in optical network optimization. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pp. 2267–2271.

Goulart, N.; de Souza, S.; Dias, L. e Noronha, T. (2011b). Biased random-key genetic algorithm for fiber installation in optical network optimization. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pp. 2267–2271.

Grasas, A.; Ramalhinho, H.; Pessoa, L. S.; Resende, M. G.; Caballé, I. e Barba, N. (2014). On the improvement of blood sample collection at clinical laboratories. *BMC Health Services Research*, 14(1):1–9.

Heilig, L.; Lalla-Ruiz, E. e Voß, S. (2015). *A Biased Random-Key Genetic Algorithm for the Cloud Resource Management Problem*, pp. 1–12. Springer International Publishing, Cham.

Hokama, P.; San Felice, M. C.; Bracht, E. C. e Usberti, F. L. (2014). A heuristic approach for the stochastic steiner tree problem. *11th DIMACS Implementation Challenge*.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. The University of Michigan Press.

Kirke, T.; While, L. e Kendall, G. (2013). Multi-drop container loading using a multi-objective evolutionary algorithm. In *2013 IEEE Congress on Evolutionary Computation*, pp. 165–172.

Lalla-Ruiz, E.; González-Velarde, J. L.; Melián-Batista, B. e Moreno-Vega, J. M. (2014). Biased random key genetic algorithm for the tactical berth allocation problem. *Appl. Soft Comput.*, 22:60–76.

Laoufi, A.; Hadjeri, S. e Hazzab, A. (2006). Adaptive probabilities of crossover and mutation in genetic algorithms for power economic dispatch. *International Journal of Applied Engineering Research*, 1(3):393–408.

López-Ibáñez, M.; Dubois-Lacoste, J.; Cáceres, L. P.; Birattari, M. e Stützle, T. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58.

Luby, M.; Sinclair, A. e Zuckerman, D. (1993). Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173 – 180.

Lucena, M. L.; Andrade, C. E.; Resende, M. G. e Miyazawa, F. K. (2014). Some extensions of biased random-key genetic algorithms. In *Anais do XLVI Simpósio Brasileiro de Pesquisa Operacional*, pp. 2469–2480.

Matsumoto, M. e Nishimura, T. (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30.

Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA.

Morán-Mirabal, L. F.; González-Velarde, J. L. e Resende, M. G. C. (2013a). *Hybrid Metaheuristics: 8th International Workshop, HM 2013, Ischia, Italy, May 23-25, 2013. Proceedings*, chapter Automatic Tuning of GRASP with Evolutionary Path-Relinking, pp. 62–77. Springer Berlin Heidelberg, Berlin, Heidelberg.

Morán-Mirabal, L. F.; González-Velarde, J. L. e Resende, M. G. C. (2013b). Randomized heuristics for the family traveling salesperson problem. In *International Transactions in Operational Research*.

Morán-Mirabal, L. F.; González-Velarde, J. L.; Resende, M. G. C. e Silva, R. M. A. (2013c). Randomized heuristics for handover minimization in mobility networks. *Journal of Heuristics*, 19(6):845–880.

Moreira, M. C. O.; Ritt, M.; Costa, A. M. e Chaves, A. A. (2012). Simple heuristics for the assembly line worker assignment and balancing problem. *Journal of Heuristics*, 18(3):505–524.

Noronha, T. F.; Resende, M. G. e Ribeiro, C. C. (2011). A biased random-key genetic algorithm for routing and wavelength assignment. *J. of Global Optimization*, 50(3).

OpenMP (2017). OpenMP API specification for parallel programming. `http://www.openmp.org`. Acessado em 22/11/2017.

Papadimitriou, C. H. e Steiglitz, K. (1982). *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Pedrola, O.; Careglio, D.; Klinkowski, M.; Velasco, L.; Bergman, K. e Solé-Pareta, J. (2013a). Metaheuristic hybridizations for the regenerator placement and dimensioning problem in sub-wavelength switching optical networks. *European Journal of Operational Research*, 224(3):614 – 624.

Pedrola, O.; Ruiz, M.; Velasco, L.; Careglio, D.; de Dios, O. G. e Comellas, J. (2013b). A grasp with path-relinking heuristic for the survivable ip/mpls-over-wson multi-layer network optimization problem. *Computers & Operations Research*, 40(12):3174 – 3187.

Prasetyo, H.; Fauza, G.; Amer, Y. e Lee, S. (2015). Survey on applications of biased-random key genetic algorithms for solving optimization problems. In *Industrial Engineering and Engineering Management (IEEM), 2015 IEEE International Conference on*, pp. 863–870. IEEE.

Reis, R.; Ritt, M.; Buriol, L. S. e Resende, M. G. C. (2011). A biased random-key genetic algorithm for ospf and deft routing to minimize network congestion. *International Transactions in Operational Research*, 18(3):401–423.

Resende, M. G. C. (2011). Biased random-key genetic algorithms with applications in telecommunications. *TOP*, 20(1):130–153.

Resende, M. G. C.; Toso, R. F.; Gonçalves, J. F. e Silva, R. M. A. (2011). A biased random-key genetic algorithm for the steiner triple covering problem. *Optimization Letters*, 6(4):605–619.

Roque, L. A. C.; Fontes, D. B. M. M. e Fontes, F. A. C. C. (2011). *A Biased Random Key Genetic Algorithm Approach for Unit Commitment Problem*, pp. 327–339. Springer Berlin Heidelberg, Berlin, Heidelberg.

Roque, L. A. C.; Fontes, D. B. M. M. e Fontes, F. A. C. C. (2014). A hybrid biased random key genetic algorithm approach for the unit commitment problem. *Journal of Combinatorial Optimization*, 28(1):140–166.

Ruiz, E.; Albareda-Sambola, M.; Fernández, E. e Resende, M. G. (2015). A biased random-key genetic algorithm for the capacitated minimum spanning tree problem. *Computers & Operations Research*, 57:95 – 108.

Ruiz, M.; Pedrola, O.; Velasco, L.; Careglio, D.; Fernández-Palacios, J. e Junyent, G. (2011). Survivable ip/mpls-over-wson multilayer network optimization. *Optical Communications and Networking, IEEE/OSA Journal of*, 3(8):629–640.

Silva, R.; Resende, M. e Pardalos, P. (2015). A python/c++ library for bound-constrained global optimization using a biased random-key genetic algorithm. *Journal of Combinatorial Optimization*, 30(3):710–728.

Silva, R.; Resende, M.; Pardalos, P. e Gonçalves, J. (2012). Biased random-key genetic algorithm for bound-constrained global optimization. In *Proceedings of the global optimization workshop*, pp. 133–136.

Silva, R. M.; Resende, M. G. e Pardalos, P. M. (2014). Finding multiple roots of a box-constrained system of nonlinear equations with a biased random-key genetic algorithm. *Journal of Global Optimization*, 60(2):289–306.

Silva, R. M. A.; Resende, M. G. C.; Pardalos, P. M. e Facó, J. L. (2013a). Biased random-key genetic algorithm for nonlinearly-constrained global optimization. In *2013 IEEE Congress on Evolutionary Computation*, pp. 2201–2206.

Silva, R. M. d. A.; Resende, M. G.; Pardalos, P. M. e Faco, J. L. (2013b). Biased random-key genetic algorithm for linearly-constrained global optimization. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '13 Companion, pp. 79–80, New York, NY, USA. ACM.

Spears, V. M. e Jong, K. A. D. (1991). On the virtues of parameterized uniform crossover. In *In Proceedings of the Fourth International Conference on Genetic Algorithms*, pp. 230–236.

Stefanello, F.; Buriol, L. S.; Hirsch, M. J.; Pardalos, P. M.; Querido, T.; Resende, M. G. C. e Ritt, M. (2015). On the minimization of traffic congestion in road networks with tolls. *Annals of Operations Research*, pp. 1–21.

Stefanello, F.; Buriol, L. S. e Resende, M. G. C. (2013). A biased random-key genetic algorithm for a network pricing problem. In *Anais do XLV Simpósio Brasileiro de Pesquisa Operacional*, p. 12, Natal - RN, Brazil. Sociedade Brasileira de Pesquisa Operacional.

Stroustrup, B. (2000). *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edição.

Tangpattanakul, P.; Jozefowiez, N. e Lopez, P. (2012). *Parallel Problem Solving from Nature - PPSN XII: 12th International Conference, Taormina, Italy, September 1-5, 2012, Proceedings, Part II*, chapter Multi-objective Optimization for Selecting and Scheduling Observations by Agile Earth Observing Satellites, pp. 112–121. Springer Berlin Heidelberg, Berlin, Heidelberg.

Tangpattanakul, P.; Jozefowiez, N. e Lopez, P. (2015). *Biased Random Key Genetic Algorithm forÂ Multi-user Earth Observation Scheduling*, pp. 143–160. Springer International Publishing, Cham.

Thomas, J. e Chaudhari, N. S. (2014). A new metaheuristic genetic-based placement algorithm for 2d strip packing. *Journal of Industrial Engineering International*, 10(1):47.

Toso, R. F. e Resende, M. G. C. (2014). A c++ application programming interface for biased random-key genetic algorithms. *Optimization Methods and Software*.

Zheng, J.-N.; Chien, C.-F. e Gen, M. (2015). Multi-objective multi-population biased random-key genetic algorithm for the 3-d container loading problem. *Computers & Industrial Engineering*, 89:80 – 87. Maritime logistics and transportation intelligence.

Certifico que o aluno **Rafael Louback Ferraz,** autor do trabalho de conclusão de curso intitulado "**Extensão da Interface de Programação de Aplicações do Algoritmo Genético de Chaves Aleatórias Viciadas**"**,** efetuou as correções sugeridas pela banca examinadora e que estou de acordo com a versão final do trabalho.

Marco Antonio Moreira de Carvalho
Orientador

Ouro Preto, 20 de dezembro de 2018.