DISSERTAÇÃO DE MESTRADO Nº 810

ALGORITMO BASEADO EM EVOLUÇÃO DIFERENCIAL PARA SOLUÇÃO DE PROBLEMAS DE OTIMIZAÇÃO COMBINATÓRIA

André Luiz Maravilha Silva

DATA DA DEFESA: 28/02/2014

Universidade Federal de Minas Gerais

Escola de Engenharia

Programa de Pós-Graduação em Engenharia Elétrica

ALGORITMO BASEADO EM EVOLUÇÃO DIFERENCIAL PARA SOLUÇÃO DE PROBLEMAS DE OTIMIZAÇÃO COMBINATÓRIA

André Luiz Maravilha Silva

Dissertação de Mestrado submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Escola de Engenharia da Universidade Federal de Minas Gerais, como requisito para obtenção do Título de Mestre em Engenharia Elétrica.

Orientador: Prof. Felipe Campelo França Pinto
Co-orientador: Prof. Jaime Arturo Ramírez

Belo Horizonte - MG

Fevereiro de 2014

Silva, André Luiz Maravilha.
S586a Algorítmo baseado em evolução diferencial para solução de problemas de otimização combinatória [manuscrito] / André Luiz Maravilha Silva. – 2014.
xx, 66 f., enc.: il.

Orientador: Felipe Campelo França Pinto.
Coorientador: Jaime Arturo Ramírez.

Dissertação (mestrado) Universidade Federal de Minas Gerais, Escola de Engenharia.

Bibliografia: f. 61-66.

1. Engenharia elétrica - Teses. 2. Otimização combinatória - Teses. 3. Heurística - Teses. I. Pinto, Felipe Campelo França. II. Ramírez, Jaime Arturo. III. Universidade Federal de Minas Gerais, Escola de Engenharia. IV. Título.

CDU: 621.3(043)

"Algoritmo Baseado em Evolução Diferencial para Solução de Problemas de Otimização Combinatória"

André Luiz Maravilha Silva

Dissertação de Mestrado submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Escola de Engenharia da Universidade Federal de Minas Gerais, como requisito para obtenção do grau de Mestre em Engenharia Elétrica.

Aprovada em 28 de fevereiro de 2014.

Por:

Prof. Dr. Felipe Campelo França Pinto
DEE (UFMG) - Orientador

Prof. Dr. Jaime Arturo Ramírez
DEE (UFMG)

Prof. Dr. Eduardo Gontijo Carrano
DEE (UFMG)

Prof. Dr. Ricardo Hiroshi Caldeira Takahashi
DMAT (UFMG)

Prof. Dr. Martin Gómez Ravetti
DEP (UFMG)

Resumo

Problemas de otimização combinatória são definidos sobre conjuntos e a solução destes problemas pode ser entendida como a definição de um subconjunto de possíveis elementos de forma a otimizar uma função objetivo, sujeito a restrições. Problemas desta natureza podem ser identificados em inúmeras situações reais, porém, apesar de serem de simples entendimento, encontrar a solução ótima pode ser uma tarefa inviável. Muitos problemas de otimização combinatória pertencem à classe de problemas $\mathcal{NP}$-difícil. Assim, o estudo e desenvolvimento de novas técnicas algorítmicas para obtenção de boas soluções é muito importante para esta área. Um algoritmo que tem atraído a atenção de pesquisadores é o algoritmo de evolução diferencial (DE, do inglês *differential evolution*) por apresentar uma boa capacidade de convergência e relativa simplicidade de implementação e compreensão. Porém, o DE é um algoritmo que foi originalmente projetado para solução de problemas de otimização de variáveis contínuas. Devido às suas características, alguns pesquisadores têm tentado adaptar este algoritmo para a solução de problemas de otimização combinatória. No entanto, tais adaptações não preservam as características que atraíram a atenção ao DE original, e o comportamento destas adaptações não vai muito além de uma busca aleatória no espaço de soluções. Acredita-se que isso ocorre devido à uma escolha inadequada para codificação das soluções. Diante disso, o presente trabalho adota uma codificação baseada em conjuntos para uso com a estrutura do DE. Além disso, os operadores aritméticos da mutação diferencial são substituídos por operações sobre conjuntos e, ainda assim, mantendo suas características. Experimentos computacionais sugerem a superioridade da técnica proposta em relação a outras adaptações existentes do DE, utilizando como base o problema do caixeiro viajante. Além disso, a técnica proposta foi comparada com outras abordagens para a solução de problemas de otimização combinatória, retornando resultados competitivos em relação aos demais métodos para o problema de agrupamento centrado capacitado.

Palavras-chave: Otimização Combinatória, Heurísticas, Metaheurísticas, Evolução Diferencial, Abordagem Baseada em Conjuntos.

Abstract

Combinatorial optimization problems are defined on sets and the solution for these problems can be seen as choosing a subset of possible elements to optimize an objective function, subject to some constraints. Problems of this nature can be found in many real situations but, despite being simple to understand, the task of finding optimal solution can be prohibitive. Many combinatorial optimization problems belongs to the class $\mathcal{NP}$-hard. Thus, the study and development of new algorithmic techniques to obtain good solutions is very important. An algorithm that has attracted the attention of researches is the differential evolution (DE) by its good convergence characteristics, and also for its simplicity of implementation. However, the DE was originally designed to solve continuous optimization problems. Due to its features, some researchers have attempted to adapt this algorithm for solving combinatorial optimization problems. However, these adaptations do not preserve the features that has attracted the attention to the original DE, and their behavior has been found to essentially perform little more than a random search. This is due to an inappropriate choice for encoding solutions. To address this issues we adopt an set-based approach for use with the structure of DE algorithm. The arithmetic operators of the differential mutation are replaced by operations on sets and still maintaining its features. Computational experiments suggest the superiority of the proposed technique over existing DE adaptations for combinatorial optimization, using instances of the traveling salesman problem as a testbed. The proposed adaptation was also compared to other usual approaches for combinatorial optimization, and returned competitive results for the capacitated centered clustering problem.

Keywords: Combinatorial Optimization, Heuristics, Metaheuristics, Differential Evolution, Set-Based Approach.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

CAPÍTULO 1

Introdução

Devido às suas inúmeras aplicações práticas, a otimização combinatória é um dos tópicos mais ativos da programação matemática e pesquisa operacional. Diversos problemas que vão desde a engenharia e ciência da computação à economia e biologia podem ser modelados como problemas de otimização combinatória (Du e Pardalos, 1999). Alguns exemplos são: o projeto de redes, projeto de circuitos eletrônicos, definição de quadro de horários, atribuição de frequências a telefones celulares, planejamento de produção e distribuição.

Apesar de muitos dos problemas de otimização combinatória serem de fácil entendimento, encontrar a solução ótima pode ser uma tarefa muito difícil. Muitos problemas de otimização combinatória pertencem ao conjunto de problemas denominados $\mathcal{NP}$-difíceis (Wolsey, 1998). Para tais problemas, não é conhecido nenhum método determinístico capaz de encontrar a solução ótima com complexidade de tempo polinomial. De fato, grande parte dos problemas reais modelados como problemas de otimização combinatória são problemas $\mathcal{NP}$-difíceis (Caserta e Voß, 2010).

Para esta classe de problemas o número de soluções candidatas tende a crescer exponencialmente em função da dimensão do problema, assim, enumerar todas as soluções é uma estratégia inviável para resolver problemas de grande porte, fazendo necessário o uso de técnicas mais inteligentes, como o algoritmo de *branch-and-bound* (Wolsey, 1998). No entanto, mesmo com o uso destes algoritmos mais inteligentes, encontrar a solução ótima pode demandar um tempo inviável, principalmente em problema combinatórios que envolvam uma função objetivo ou alguma restrição não-linear, pois tal características viola as premissas de convergência destes algoritmos (Belotti *et al.*, 2013).

Com isso, o uso de heurísticas e metaheurísticas na solução de problemas de otimização combinatória é comum e bastante aceito na área da pesquisa operacional. Tais métodos abrem mão da garantia de otimalidade por estratégias que encontrem soluções sub-ótimas em um tempo viável. Assim, a pesquisa e desenvolvimento de heurísticas e metaheurísticas eficientes é de grande importância.

Uma metaheurística que tem atraído a atenção de diversos pesquisadores de otimização dos mais variados domínios, inclusive da otimização combinatória, é o algoritmo de evolução diferencial (DE, do inglês *differential evolution*), introduzido por Storn & Price em meados dos anos 90 (Storn e Price, 1995, 1997). O DE é um algoritmo evolucionário para otimização de funções, podendo ser funções não-lineares e não-diferenciáveis, no domínio de variáveis contínuas.

Este algoritmo apresenta uma boa capacidade de convergência para regiões próximas do ótimo global, muitas vezes encontrando a solução ótima para diversos problemas-teste comumente utilizados na avaliação de métodos de otimização de funções de variáveis contínuas (Storn e Price, 1997). O DE apresenta uma precisão e velocidade de convergência superiores a muitos outros métodos de otimização, o que o levou conquistar as primeiras colocações do IEEE *International Contest on Evolutionary Optimization* nos anos de 1996 e 1997.

Devido às boas características de convergência e qualidade da solução retornada na otimização de funções no domínio de variáveis contínuas, alguns pesquisadores têm adaptado o DE para solucionar problemas de otimização combinatória. No entanto, as adaptações encontradas não preservam estas boas características que atraíram sua atenção (Storn, 2008). A maior parte destas adaptações são limitadas à resolução de problemas de otimização combinatória baseados em permutação, não sendo aplicáveis a problemas de otimização combinatória de maneira geral (Prado *et al.*, 2010).

Além disso, muitas das adaptações utilizam uma codificação de soluções que impossibilita o DE trabalhar com os dados do problema que contenham as informações realmente relevantes. Com isso, as adaptações atuais exploram o espaço de busca fazendo pouco mais que uma busca aleatória. Assim, faz-se necessário a utilização de uma codificação de soluções que possibilite ao DE e algoritmos semelhantes, manipular os dados do problema diretamente em sua codificação, realizando assim a exploração do espaço de busca com alguma inteligência embutida.

Uma vez que problemas de otimização combinatória consistem principalmente em explorar um espaço de busca que pode ser descrito por um conjunto, conjuntos podem ser considerados uma representação natural para esta classe de problemas. Por exemplo, em problemas de grafo as soluções são diretamente representadas como um conjunto de arestas, similar à abordagem *edge-set* utilizada para a otimização de estruturas em árvore (Raidl e Julstrom, 2003; Rothlauf e Tzschoppe, 2005).

1.1 Objetivos

O objetivo geral deste trabalho é utilizar uma codificação baseada em conjunto juntamente com a estrutura do DE para resolver problemas de otimização combinatória de maneira geral, possibilitando ao algoritmo manipular as informações relevantes do problema a ser resolvido.

A fim de atingir tal objetivo, são definidos os seguintes objetivos específicos:

- apresentar e explorar as características de conjuntos para que possam ser utilizados na codificação de problemas combinatórios de maneira geral;
- adaptar os operadores de mutação e recombinação do algoritmo de evolução diferencial para que possam trabalhar com uma codificação baseada em conjuntos, preservando suas características;
- propor um algoritmo baseado no algoritmo de evolução diferencial para solução de problemas de otimização combinatória;
- avaliar o comportamento do algoritmo proposto, verificando se as características do DE original são preservadas;
- avaliar a capacidade de convergência do algoritmo proposto;
- definir um planejamento experimental para avaliar o desempenho do algoritmo proposto em relação às outras adaptações do DE e outras abordagens para solução de problemas de otimização combinatória.

1.2 Estrutura do trabalho

Os capítulos restantes deste trabalho estão organizados da seguinte forma:

No Capítulo 2 são apresentados alguns conceitos de problemas de otimização combinatória, sua relação com problemas de programação matemática, e também sua classificação em problemas de otimização baseados em permutação e não-baseados em permutação. Após a apresentação destes conceitos, são vistas as metodologias comuns para solução desta classe

de problemas e, por fim, as adaptações do algoritmo de evolução diferencial existentes para resolver problemas de natureza combinatória.

No Capítulo 3 são apresentadas a codificação baseada em conjuntos para representação de soluções de problemas de otimização combinatória e a sua integração com o algoritmo de evolução diferencial para solução desta classe de problemas. Nele são descritas ainda as modificações nos operadores de mutação e recombinação, possibilitando a manipulação de conjuntos ao invés de valores numéricos. Após a descrição do algoritmo, é analisado o seu comportamento ao longo das iterações na solução de uma instância do problema do caixeiro viajante, mostrando a capacidade do algoritmo de convergir em direção a uma única solução, como ocorre com o algoritmo de evolução diferencial original.

No Capitulo 4 são descritos os experimentos computacionais realizados para avaliação do desempenho do algoritmo proposto, comparando os resultados obtidos em relação às outras adaptações existentes do algoritmo de evolução diferencial e também em relação a outros métodos comuns para solução de problemas de otimização combinatória. São considerados dois problemas clássicos de otimização combinatória para realização dos experimentos: o problema do caixeiro viajante e o problema de agrupamento centrado capacitado. Além disso, é comparado também a qualidade das soluções obtidas em relação à solução ótima para os problemas em que este resultado é conhecido.

No Capítulo 5 são feitas as conclusões e são apresentados possíveis estudos futuros que podem ser realizados com o objetivo de dar continuidade ao que já foi alcançado com a realização do presente trabalho.

CAPÍTULO 2

Referencial Teórico

Neste capítulo são apresentadas a definição de um problema de otimização combinatória e sua relação com problemas de programação inteira. Em seguida, são apresentadas as metodologias mais comuns para solução desta classe de problemas. Por fim, é apresentado o algoritmo de evolução diferencial e suas adaptações existentes para solucionar problemas de otimização combinatória, bem como uma discussão sobre as principais limitações destas adaptações.

2.1 Otimização combinatória

Dado um conjunto finito e discreto $\mathbb{E} = \{e_1, e_2, \ldots, e_n\}$, um problema de otimização combinatória é definido sobre todos os subconjuntos viáveis de $\mathbb{E}$, denotado por $\mathcal{F}$ (Wolsey, 1998). Dessa forma, o conjunto de soluções viáveis $\mathcal{F}$ é um conjunto finito e discreto, ou seja, é um conjunto enumerável. Considerando uma função objetivo $f(\cdot) \ : \ \mathcal{F} \mapsto \mathbb{R}$ e uma solução $\mathbb{S} \subseteq \mathbb{E}$, um problema de otimização combinatória pode ser escrito como:

$$\text{Encontrar } \mathbb{S}^* = \arg\min_{\mathbb{S}\subseteq\mathbb{E}} \{ f(\mathbb{S}) \ : \ \mathbb{S} \in \mathcal{F} \} \ . \tag{2.1}$$

Alguns problemas de otimização combinatória podem ser escritos como problemas de programação linear inteira. Um problema de programação linear inteira é um problema da forma:

$$\begin{aligned} \min \quad & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{sujeito a:} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{G}\mathbf{x} = \mathbf{h} \\ & \mathbf{x} \geq \mathbf{0} \text{ e inteiro,} \end{aligned} \tag{2.2}$$

onde $\mathbf{x}$ é o vetor de variáveis de decisão, $\mathbf{c}$ é o vetor de custos, $\mathbf{A}$ é a matriz de coeficientes das restrições de desigualdade, $\mathbf{G}$ é a matriz de coeficientes das restrições de igualdade e, $\mathbf{b}$ e $\mathbf{h}$ são

os vetores de demanda associados às restrições de desigualdade e igualdade, respectivamente (Bertsimas e Tsitsiklis, 1997).

Para escrever um problema de otimização combinatória como um problema de programação linear inteira, são empregadas variáveis de decisão $x_i \in \{0,1\}$ para cada elemento $e_i \in \mathbb{E}$. Estas variáveis codificam uma escolha entre duas alternativas (Bertsimas e Tsitsiklis, 1997). Dessa forma, a variável x_i assume o valor 1 quando seu respectivo elemento e_i compõe a solução para o problema, 0 (zero) caso contrário.

Através das variáveis x_i deve-se definir um conjunto de restrições de forma que as soluções viáveis do modelo matemático correspondam ao conjunto de subconjuntos válidos do problema original. Além disso, a função objetivo deve ser reescrita em função das variáveis x_i. Ao escrever um problema de otimização combinatória como um problema de programação linear inteira, o conjunto de possíveis soluções viáveis é tratado implicitamente (Wolsey, 1998).

Como os possíveis valores das variáveis x_i são limitados ao conjunto $\{0,1\}$, esses problemas são também denominados de problemas de programação inteira binária ou ainda, problemas de programação inteira 0-1 (Rao, 2009; Wolsey, 1998).

Um exemplo de problema de otimização combinatória é o problema da mochila 0-1 (Martello e Toth, 1990). Esse problema é definido por uma mochila de capacidade K e um conjunto de itens $\mathbb{E} = \{e_1, e_2, \ldots, e_n\}$, onde cada item e_i possui um benefício p_i e um peso w_i. O objetivo é determinar um subconjunto $\mathbb{S} \subseteq \mathbb{E}$ que maximize o benefício total obtido pela soma dos item $e_i \in \mathbb{S}$ sem que a soma de seus pesos extrapole a capacidade máxima da mochila. Assim, este problema pode ser escrito como:

$$\text{Encontrar } \mathbb{S}^* = \arg\max_{\mathbb{S}\subseteq\mathbb{E}} \left\{ \sum_{e_i \in \mathbb{S}} p_i \ : \ \mathbb{S} \in \mathcal{F} \right\}, \tag{2.3}$$

onde $\mathcal{F} = \{\mathbb{S} \subseteq \mathbb{E} \ : \ \sum_{e_i \in \mathbb{S}} w_i \leq K\}$, ou seja, o conjunto de todos os subconjuntos de itens em $\mathbb{E}$ tais que a soma de seus pesos não seja maior que a capacidade K da mochila. Este mesmo

problema pode ser escrito como um problema de programação inteira binária da forma:

$$\max \sum_{i=1}^{n} p_i x_i \tag{2.4}$$

$$\sum_{i=1}^{n} w_i x_i \leq K \tag{2.5}$$

$$\mathbf{x} = \{0,1\}^n \tag{2.6}$$

onde a função objetivo (2.4) é a soma dos benefícios dos itens dada pela solução $\mathbf{x}$, a restrição (2.5) elimina as soluções em que a soma dos pesos de seus itens extrapola a capacidade máxima da mochila e a restrição (2.6) define o domínio das variáveis de decisão.

2.1.1 Problemas de otimização combinatória baseados em permutação

Problemas de otimização combinatória também podem ser definidos a partir de problemas de permutação. Tais problemas são denominados problemas de otimização combinatória baseados em permutação (Onwubolu e Davendra, 2009b).

Dado um conjunto finito e discreto de elementos $\mathbb{E} = \{e_1, e_1, \ldots, e_n\}$, o espaço de busca em um problema de permutação é dado por todas as permutações válidas entre os elementos de $\mathbb{E}$. Caso existam restrições, algumas permutações podem definir soluções inválidas. Portanto, assim como problemas de otimização combinatória, o espaço de busca é dado por um conjunto enumerável de soluções (Onwubolu e Davendra, 2009b; Price *et al.*, 2005).

Um exemplo de problema de otimização combinatória baseada em permutação é o problema do caixeiro viajante (TSP, do inglês *traveling salesman problem*) (Garfinkel, 1985; Wolsey, 1998). Dado $\mathbb{E} = \{e_1, e_1, \ldots, e_n\}$ como o conjunto de cidades e $c_{ij} \geq 0$ como a distância que deve ser percorrida para alcançar a cidade e_j imediatamente após a cidade e_i, o objetivo é determinar o percurso de menor distância que passe por todas as cidades, retornando à cidade de origem no final do trajeto. Assim, uma solução para esse problema é definida por uma permutação entre as cidades.

Problemas dessa natureza podem ser modelados por um grafo $\mathcal{G} = (\mathbb{V}, \mathbb{A})$, onde $\mathbb{V}$ é o conjunto de vértices do grafo e $\mathbb{A}$ é o conjunto de arestas que conectam pares de vértices. Para cada elemento do problema de permutação existe um vértice no grafo, ou seja, $\mathbb{V} = \{v_i : \exists e_i \in \mathbb{E}\}$.

As arestas do grafo representam as regras de precedência válidas para o problema. Assim uma aresta $(v_i, v_j) \in \mathbb{A}$ conectando o vértice v_i ao vértice v_j indica que o elemento e_j pode ocorrer imediatamente após o elemento e_i em uma permutação.

Uma solução para um problema de otimização combinatória baseada em permutação pode ter suas soluções representadas não apenas por uma permutação dos elementos, mas também pelo conjunto das arestas que definem esta permutação. Dessa forma, ao invés de definir o problema como uma permutação dos elementos $\mathbb{E} = \{e_1, e_2, \ldots, e_n\}$, ele pode ser definido como um problema de otimização combinatória sobre o conjunto de arestas que representam as regras de precedência. Considerando e_{ij} como a aresta (v_i, v_j), o conjunto de elementos para o problema de otimização passa a ser $\mathbb{E}' = \{e_{ij} \ : \ \exists (v_i, v_j) \in \mathbb{A}\}$.

Assim, o problema do caixeiro viajante também pode ser escrito como um problema de programação inteira binária da forma:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{2.7}$$

$$\sum_{j:j\neq i} x_{ij} = 1 \qquad \forall\, i = 1, \ldots, n \tag{2.8}$$

$$\sum_{i:i\neq j} x_{ij} = 1 \qquad \forall\, j = 1, \ldots, n \tag{2.9}$$

$$\sum_{e_i \in \mathbb{S}} \sum_{e_j \in \mathbb{S}} x_{ij} \leq |\mathbb{S}| - 1 \qquad \mathbb{S} \subset \mathbb{E},\ 2 \leq |\mathbb{S}| \leq n-1 \tag{2.10}$$

$$\mathbf{x} = \{0,1\}^n \tag{2.11}$$

onde as restrições (2.8) e (2.9) definem que para cada vértice, a solução terá uma única aresta chegando e uma única aresta saído, respectivamente. A restrição (2.10) é chamada de eliminação de subciclos, que garante que a solução será composta por um único ciclo. E, por fim, a restrição (2.11) define o domínio das variáveis de decisão.

2.2 Metodologias para solução de problemas de otimização combinatória

Como o espaço de soluções em problemas de otimização combinatória é um conjunto enumerável, uma estratégia simples para resolver problemas desta classe é enumerar e avaliar

todas as possíveis soluções e assim, determinar qual é a solução ótima.

Considerando o problema da mochila 0-1, existem 2^n subconjuntos de $\mathbb{E}$ possíveis. Dado que a capacidade da mochila é $K = \sum_{i=1}^{n} w_i/2$, pelo menos metade dos subconjuntos são viáveis, assim, existem no mínimo 2^{n-1} soluções válidas (Wolsey, 1998).

Para o problema do caixeiro viajante existem $n!$ permutações possíveis. Porém, como vetores rotacionados representam uma mesma solução, existem $(n-1)!$ soluções diferentes e válidas. Para uma instância do TSP de $n = 101$ cidades, o número de soluções válidas é aproximadamente $9,33 \times 10^{157}$.

Este rápido crescimento do número de possíveis soluções é denominado *explosão combinatória*. Portanto, enumerar todas as possíveis soluções para resolver problemas de otimização combinatória é viável apenas para problemas de pequena dimensão. Com isso, são necessárias estratégias mais inteligentes para resolver problemas de grandes dimensões.

2.2.1 Algoritmos exatos

Os algoritmos exatos garantem a obtenção a solução ótima para problemas combinatórios. Geralmente, estes algoritmos definem estratégias iterativas que resolvem subproblemas obtidos a partir de relaxações do problema original, limitando o espaço busca a cada iteração, até que a solução ótima seja encontrada.

Um algoritmo exato e muito conhecido é o algoritmo *branch-and-bound*. Este algoritmo utiliza uma estratégia de dividir para conquistar, criando uma árvore de enumeração das soluções de maneira inteligente e iterativa. A cada iteração, o *branch-and-bound* expande a árvore de enumeração particionando o espaço de soluções, criando problemas mais restritos. Através de limites calculados ao longo da enumeração ele é capaz de identificar nós da árvore que não levarão à solução ótima, assim evitando a exploração de todo o espaço de soluções (Goldbarg e Luna, 2005; Wolsey, 1998).

Existem também outras técnicas para solucionar problemas de otimização combinatória além do *branch-and-bound*, como os algoritmos de planos de corte (Wolsey, 1998), algoritmos de geração de colunas (Lübbecke e Desrosiers, 2005; Wolsey, 1998) e vários outros. Os algoritmos de planos de corte e geração de colunas podem ser também combinados ao *branch-and-bound*, gerando assim os algoritmos *branch-and-cut* e *branch-and-price*, respectivamente (Wolsey, 1998).

Existe uma classe de problemas de otimização para os quais não é conhecido nenhum algoritmo, em máquina determinística, com complexidade de tempo polinomial que garanta encontrar a solução ótima. Esta classe é denominada $\mathcal{NP}$-difícil (Wolsey, 1998). Muitas situações reais, quando modeladas como problemas de otimização, pertencem a esta classe de problemas. Assim, apesar dos algoritmos exatos garantirem a solução ótima, eles podem demandar um número exponencial de iterações, tornando-se uma estratégia inviável para problemas de grandes dimensões (Bertsimas e Tsitsiklis, 1997).

Além da falta de métodos exatos eficientes que resolvam tais problemas em tempo razoável, existem outras razões que podem impossibilitar a utilização de métodos exatos para a solução de problemas de otimização (Wolsey, 1998):

- instâncias muito grandes ou complicadas de serem modeladas como problemas de programação inteira;
- mesmo que o problema tenha sido modelado como um problema de programação inteira, métodos exatos como *branch-and-bound* podem apresentar dificuldades em encontrar soluções viáveis;
- abordagens gerais para problemas de programação inteira não são muito eficientes para certos problemas combinatórios como problemas de roteamento de veículos e escalonamento de máquinas, embora seja fácil encontrar soluções viáveis por inspeção ou conhecimento da estrutura do problema.

2.2.2 Heurísticas e Metaheurísticas

O termo heurística vem do grego *heuriskein*, que significa descobrir ou achar. Mas o significado da palavra na área da pesquisa operacional vai um pouco além de sua raiz etimológica (Goldbarg e Luna, 2005). Uma heurística pode ser definida como a parte do algoritmo de otimização que utiliza informações obtidas pelo próprio algoritmo para ajudar a decidir qual a próxima solução candidata a ser testada ou como a próxima solução deve ser construída (Weise, 2009).

A ideia principal das heurísticas é tentar encontrar soluções boas e viáveis rapidamente utilizando informações específicas do problema para o qual foram projetadas, porém, não oferecem nenhuma garantia de otimalidade (Maniezzo *et al.*, 2010). Heurísticas são específicas para o problema e, assim, frequentemente, um método que trabalha para um determinado problema

não pode ser utilizado para solucionar um outro problema diferente (Ehrgott e Gandibleux, 2000).

Alguns algoritmos constroem soluções garantidamente dentro de uma margem de erro pré-estabelecida em relação à solução ótima com complexidade de tempo polinomial, diferente das heurísticas que não dão nenhuma garantia de qualidade da solução retornada. Estes algoritmos são denominados algoritmos aproximativos (Wolsey, 1998).

Devido à dificuldade de se obter boas soluções em um tempo computacional viável através de métodos exatos, o uso de métodos heurísticos é amplamente aceito pela comunidade da pesquisa operacional (Maniezzo *et al.*, 2010; Wolsey, 1998).

Um tópico bastante explorado na área da otimização combinatória são as metaheurísticas. Diferentemente das heurísticas e algoritmos aproximativos, que definem estratégias específicas para um dado problema, as metaheurísticas definem estratégias gerais para construção de algoritmos de otimização. Em outras palavras, as metaheurísticas definem estratégias iterativas que guiam o uso de operações e outras heurísticas para produzir novas soluções de maneira eficiente, podendo estas manipular uma única solução ou várias soluções a cada iteração (Maniezzo *et al.*, 2010).

Na literatura são encontradas diversas metaheurísticas e muitos trabalhos que fazem uso dessas técnicas na construção de algoritmos para solucionar os mais variados problemas de otimização.

Para tentar evitar o problema de convergência para atratores definidos por ótimos locais, as metaheurísticas utilizam estratégias de diversificação que as permitem escapar destas regiões e assim, realizar uma busca robusta no espaço de soluções (Glover e Kochenberger, 2003).

Algumas metaheurísticas fazem uso de métodos de buscas local (Definições 2.1 e 2.2) para guiar a exploração do espaço de soluções, sendo por isso denominadas metaheurísticas baseadas em busca local.

Definição 2.1 (Função de vizinhança)**.** *Uma função de vizinhança* $\mathcal{N}(\cdot) : \mathcal{F} \mapsto 2^{\mathcal{F}}$ *faz um mapeamento de cada solução* $\mathbf{x} \in \mathcal{F}$ *para um conjunto de todas as outras soluções em* $\mathcal{F}$ *que podem ser obtidas com a aplicação de um movimento unitário de um conjunto pré-especificado (Michalewicz e Fogel, 2004).*

Definição 2.2 (Busca local)**.** *Uma busca local é um método que move iterativamente pelas soluções do conjunto* $\mathcal{F}$*, limitado pela vizinhança* $\mathcal{N}(\mathbf{x}) \subseteq \mathcal{F}$ *da solução atual* $\mathbf{x}$*. A escolha da próxima solução é baseada na solução atual e possivelmente em soluções previamente visitadas*

(Michalewicz e Fogel, 2004). O Algoritmo 2.1 apresenta a estrutura básica de um método de busca local.

Entrada: $\mathbf{x}^0$: solução inicial
Entrada: $\mathcal{N}(\cdot)$: função de vizinhança
$\mathbf{x} \leftarrow \mathbf{x}^0$;
repita
  escolher $\mathbf{x}' \in \mathcal{N}(\mathbf{x})$;
  $\mathbf{x} \leftarrow \mathbf{x}'$;
até *critério de parada ser satisfeito*;
retorna $\mathbf{x}$;

Algoritmo 2.1: Estrutura básica de um método de busca local

A principal característica que diferencia uma metaheurística de busca local de outra é a forma como uma nova solução é escolhida a cada iteração (Lust, 2010).

A metaheurística *simulated annealing* (Kirkpatrick *et al.*, 1983; Nikolaev e Jacobson, 2010) explora uma única solução a cada iteração, podendo aceitar soluções piores com determinada probabilidade como estratégia de diversificação. A metaheurística *tabu seach* (Gendreau e Potvin, 2010; Glover, 1986) se assemelha ao *simulated annealing*, porém mantém uma estrutura de memória (lista tabu) que proíbe certos movimentos que podem levar a soluções já exploradas.

O *iterated local search* (Lourenço *et al.*, 2003, 2010) é uma metaheurística que realiza, a cada iteração, uma busca local a partir da solução atual, mantendo sempre a melhor solução encontrada. Como estratégia de diversificação são utilizados mecanismos de perturbação que realizam alterações na solução atual para que uma nova busca local possa ser realizada a partir de uma outra região do espaço de busca. Já o *variable neighborhood search* (Hansen *et al.*, 2010; Mladenović e Hansen, 1997) alterna entre diversas funções de vizinhança para geração da próxima solução que será explorada.

Uma outra metaheurística muito utilizada é o *greedy random adaptative search procedure* (Resende e Ribeiro, 2010), que tem o foco na construção da solução, sendo a busca local utilizada apenas para gerar pequenas melhorias na solução construída. Este é um método considerado semi-guloso, pois a escolha do próximo elemento a fazer parte da solução é realizada aleatoriamente dentre os melhores classificados.

Algumas metaheurísticas manipulam um conjunto de soluções, chamado de população, a cada iteração. Estas metaheurísticas são denominadas metaheurísticas baseadas em popula-

ção. A população de soluções evolui a cada iteração, gerando novas soluções a partir da troca de conhecimento adquirido ao logo do processo iterativo.

Exemplos de metaheurísticas baseadas em população e bastante conhecidas são os algoritmos evolucionários (Neumann e Witt, 2010). Estes algoritmos geram novas soluções, geralmente a partir de soluções presentes na população corrente, a cada iteração e aquelas soluções de melhor qualidade tem uma probabilidade maior de permanecerem na população para a próxima iteração do algoritmo. O processo de seleção das soluções que irão compor a população para a próxima iteração faz analogia à seleção natural, onde os indivíduos mais adaptados ao ambiente tendem a viver por mais tempo.

2.3 Algoritmo de evolução diferencial

O algoritmo de evolução diferencial (DE, do inglês *differential evolution*), apresentado por Storn e Price (1995), é uma metaheurística baseada em população para otimização de funções com variáveis contínuas.

O DE é um algoritmo bastante simples mas, ainda assim, muito eficiente. Possuindo poucos parâmetros de controle e utilizando basicamente algumas operações aritméticas básicas, o DE é capaz de convergir para regiões promissoras do espaço de busca e, em muitas classes de problemas, encontrando até mesmo a solução ótima (Storn e Price, 1997).

Devido a sua simplicidade e boas características de convergência, o DE tem atraído a atenção de pesquisadores que lidam com problemas de otimização em outros domínios (Onwubolu e Davendra, 2009a). Com particular relevância para o desenvolvimento deste trabalho, estudos em que o DE é adaptado para solucionar problemas de otimização combinatória têm aparecido na literatura nos últimos anos (Onwubolu e Davendra, 2006, 2009c; Pan *et al.*, 2008; Prado *et al.*, 2010; Sauer *et al.*, 2011).

As subseções seguintes são dedicadas à apresentação do DE em sua versão original para otimização de funções com variáveis contínuas e suas adaptações para solução de problemas de otimização combinatória.

2.3.1 O algoritmo de evolução diferencial básico

O DE inicia a busca a partir de um conjunto de n_p vetores $\mathbf{x}_{i,0} \in \mathbb{R}^n$ gerados aleatoriamente, com $i \in [1, n_p]$. O algoritmo explora o espaço de busca com a criação de novos vetores a partir das operações de mutação e recombinação.

Considere o valor da j-ésima variável da i-ésima solução candidata na g-ésima iteração do algoritmo como $x^j_{i,g}$. As operações de mutação e recombinação podem ser descritas por (2.12) e (2.13), respectivamente.

$$\mathbf{v}_{i,g} = \mathbf{x}_{r_1,g} + \psi \cdot (\mathbf{x}_{r_2,g} - \mathbf{x}_{r_3,g}) \tag{2.12}$$

$$u^j_{i,g} = \begin{cases} v^j_{i,g} & \text{se } U(0,1) \leq \rho \text{ ou } j = \ell_i, \\ x^j_{i,g} & \text{caso contrário} \end{cases} \tag{2.13}$$

onde $r_1 \neq r_2 \neq r_3 \neq i \in [1, n_p]$ são índices gerados aleatoriamente, $\psi \in [0, 2]$ e $\rho \in [0, 1]$ são parâmetros definidos manualmente[1] e $\ell_i \in [1, n]$ é um inteiro gerado aleatoriamente usado para garantir que o vetor *trial* $\mathbf{u}_{i,g}$ possua no mínimo um elemento do seu vetor mutante $\mathbf{v}_{i,g}$ correspondente. A variável $U(0,1)$ indica um valor aleatório com distribuição uniforme entre 0 e 1. Soluções candidatas $\mathbf{x}_{i,g}$ são denominadas soluções *target* e $\mathbf{x}_{r_1,g}$ é denominado vetor base da operação de mutação. As Figuras 2.1 e 2.2 ilustram os procedimentos de mutação e cruzamento, respectivamente.

Após as operações de mutação e recombinação serem aplicadas a todas as soluções candidatas, uma operação de seleção determinística é utilizada para definir a população da próxima iteração:

$$\mathbf{x}_{i,g+1} = \begin{cases} \mathbf{u}_{i,g} & \text{se } f(\mathbf{u}_{i,g}) \leq f(\mathbf{x}_{i,g}), \\ \mathbf{x}_{i,g} & \text{caso contrário} \end{cases} \tag{2.14}$$

onde $f(\cdot) : \mathbb{R}^n \mapsto \mathbb{R}$ é a função objetivo que deve ser minimizada. O DE repete este processo de mutação, recombinação e seleção até que um critério de parada seja satisfeito, retornando o vetor com o menor valor de $f(\cdot)$ ao fim de sua execução. O Algoritmo 2.2 apresenta a estrutura geral do DE em sua forma básica.

Apesar de existirem outras variantes do DE descrito acima, esta é a mais utilizada para

[1] Os parâmetros de controle ψ e ρ são geralmente definidos na literatura como F e CR, respectivamente. Para manter a consistência da nomenclatura adotada ao longo do texto, optou-se por utilizar letras gregas minúsculas para definição de parâmetros de controle, enquanto letras latinas maiúsculas são utilizadas para parâmetros dos problemas de otimização e também para constantes.

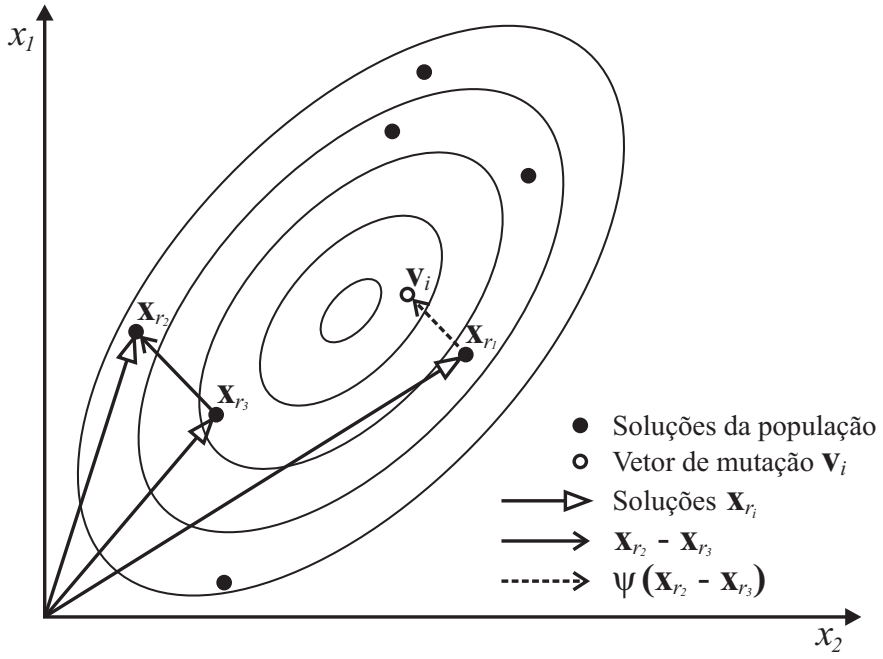


Figura 2.1 Exemplo do operador de mutação do algoritmo DE para um problema de duas variáveis. Adaptado de Storn e Price (1997).

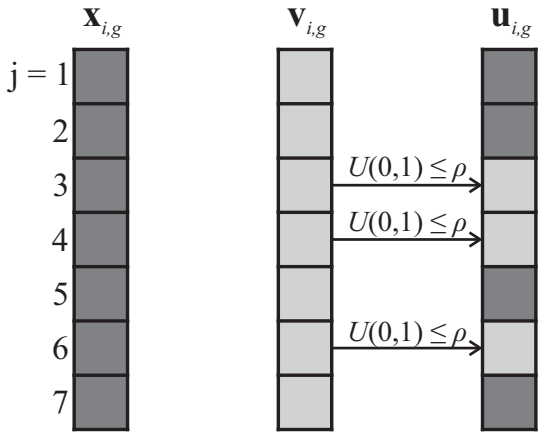


Figura 2.2 Exemplo do processo de recombinação do algoritmo DE para um problema de sete variáveis. Adaptado de (Storn e Price, 1997).

Entrada: n: número de variáveis
Entrada: $f(\cdot) : \mathbb{R}^n \mapsto \mathbb{R}$: função objetivo
Entrada: n_p: tamanho da população
Entrada: ψ: fator de escalonamento da mutação
Entrada: ρ : constante de recombinação

// Gera a população inicial
Inicializar vetores $\mathbf{x}_{1,0}, \mathbf{x}_{2,0}, \ldots, \mathbf{x}_{n_p,0} \in \mathbb{R}^n$;
$g \leftarrow 0$;
repita
  para $i \leftarrow 1$ **até** n_p **faça**
    // Mutação
    $\mathbf{v}_{i,g} \leftarrow \mathbf{x}_{r_1,g} + \psi \cdot (\mathbf{x}_{r_2,g} - \mathbf{v}_{r_3,g})$; // $r_1 \neq r_2 \neq r_3 \neq i \in [1, n_p]$
    // Recombinação
    $\mathbf{u}_{i,g} \leftarrow \mathbf{x}_{i,g}$;
    $u^{\ell}_{i,g} \leftarrow v^{\ell}_{i,g}$; // $\ell \in [1,n]$ aleatório
    para $j \leftarrow 1$ **até** n **faça**
      se $U(0,1) \leq \rho$ **então**
        $u^{j}_{i,g} \leftarrow v^{j}_{i,g}$;
      fim
    fim
    // Seleção
    se $f(\mathbf{u}_{i,g}) \leq f(\mathbf{x}_{i,g})$ **então**
      $\mathbf{x}_{i,g+1} \leftarrow \mathbf{u}_{i,g}$;
    senão
      $\mathbf{x}_{i,g+1} \leftarrow \mathbf{x}_{i,g}$;
    fim
  fim
  $g \leftarrow g + 1$;
até *critério de parada ser satisfeito*;
retorna *Melhor solução* $\mathbf{x}_{i,g}$ *encontrada*

Algoritmo 2.2: Algoritmo de evolução diferencial básico

a otimização de problemas definidos no domínio contínuo. As variantes mais usuais definem estratégias que diferem na forma como a mutação e recombinação são realizadas (Onwubolu e Davendra, 2009a; Price *et al.*, 2005).

2.3.2 Abordagens para otimização combinatória

Ao se tratar de problemas de otimização com variáveis no domínio do conjunto dos números inteiros, a estratégia mais simples para uso do DE consiste na aproximação dos valores fracionários para o inteiro mais próximo quando necessário.

Esta estratégia pode funcionar bem para muitos problemas de programação inteira e, até mesmo, para alguns problemas combinatórios. No entanto, existem problemas de otimização combinatória em que esta estratégia pode não ser viável (Prado *et al.*, 2010).

Um exemplo de problema de otimização combinatória em que esta estratégia pode ser utilizada é o problema da mochila 0-1. Associando uma variável binária $x_i \in \{0,1\}$ para cada item $e_i \in \mathbb{E}$, x_i recebe um valor igual a 1 caso o item e_i faça parte da solução ou 0 (zero) caso contrário.

Após aplicar as etapas de mutação e recombinação, os valores fracionários são arredondados para o inteiro mais próximo dentro do domínio $\{0,1\}$. Para o problema da mochila 0-1, estes valores além de indicar se um elemento compõe ou não uma solução, também podem ser vistos como a quantidade de cada item na mochila, ou seja, esses valores possuem significado numérico (Prado *et al.*, 2010).

Para problemas de otimização combinatória baseados em permutação essa estratégia não é tão simples de ser aplicada. A representação mais comum para soluções desta classe de problemas é através de um vetor $\mathbf{x}$ contendo uma permutação de números inteiros. Dessa forma, $x_i = j$ significa que e_j é o i-ésimo elemento na permutação. Esses valores inteiros atribuídos às variáveis x_i são apenas rótulos definidos arbitrariamente, não possuindo nenhum significado numérico. Com isso, as características numéricas do DE que o tornam uma poderosa técnica de otimização não são preservadas (Storn, 2008).

Como os valores são apenas rótulos, os resultados gerados pelos operadores aritméticos na mutação (2.12) não possuem nenhum significado. Além disso, mesmo utilizando a estratégia de arredondamento, os resultados gerados podem resultar em permutações inválidas, com valores repetidos, faltantes e até mesmo valores não pertencentes ao intervalo de valores

inteiros válidos.

Para ilustrar essa situação, considere o problema do caixeiro viajante com cinco cidades, rotuladas com valores de 1 a 5, e a representação de soluções por vetores de permutação de inteiros descrita acima. Dados os vetores $\mathbf{x}_{r_1,g} = (2,1,5,4,3)^{\mathrm{T}}$, $\mathbf{x}_{r_2,g} = (1,5,4,2,3)^{\mathrm{T}}$ e $\mathbf{x}_{r_3,g} = (5,3,1,4,2)^{\mathrm{T}}$, e o fator de ponderação $\psi = 0.5$, o uso do operador de mutação do DE básico resulta no vetor $\mathbf{v}_{i,g} = (0,2,6.5,3,3.5)^{\mathrm{T}}$. Após o arredondamento dos valores, é obtido o vetor de inteiros $\mathbf{v}'_{i,g} = (0,2,7,3,4)^{\mathrm{T}}$. Como pode ser observado, esta permutação não representa uma solução válida, pois os rótulos 0 e 7 não são valores válidos. Assim, a simples estratégia de arredondar os valores fracionários para o inteiro mais próximo não pode ser aplicada de maneira direta.

Diante dessa situação, alguns pesquisadores vêm apresentando estratégias para lidar com esta dificuldade do DE em resolver problemas de otimização combinatória baseados em permutação. Estes trabalhos basicamente adaptam os operadores de mutação para que as soluções resultantes sejam permutações válidas. Algumas dessas estratégias são apresentadas a seguir.

2.3.2.1 Abordagem por matriz de permutação

Na abordagem por matriz de permutação, apresentada por Price *et al.* (2005), as soluções são representadas por vetores de permutação de inteiros. A partir desta representação, a diferença entre duas soluções candidatas $\mathbf{x}_{r_2,g}$ e $\mathbf{x}_{r_3,g}$ é definida pela relação:

$$\mathbf{x}_{r_2,g} = \mathbf{P}\mathbf{x}_{r_3,g} \ , \tag{2.15}$$

onde $\mathbf{P}$, denominada matriz de permutação, mapeia o vetor $\mathbf{x}_{r_3,g}$ em outro vetor $\mathbf{x}_{r_2,g}$. Essa matriz pode ser interpretada como um conjunto de movimentos de troca que, quando aplicados em $\mathbf{x}_{r_3,g}$ se obtém $\mathbf{x}_{r_2,g}$. Assim, a partir de $\mathbf{P}$, o operador de mutação diferencial passa a ser definido por:

$$\mathbf{v}_{i,g} = \mathbf{P}\mathbf{x}_{r_1,g} \ , \tag{2.16}$$

onde os movimentos de troca definidos por $\mathbf{P}$, que levam $\mathbf{x}_{r_3,g}$ a $\mathbf{x}_{r_2,g}$, são aplicados a uma terceira solução candidata, a solução base $\mathbf{x}_{r_1,g}$.

A matriz de permutação não define o processo de recombinação, que geralmente não é utilizada nesta abordagem. Com isso, a solução obtida pela mutação diferencial é comparada

com a solução *target* $\mathbf{x}_{i,g}$ e, a melhor solução deverá compor a população da próxima iteração.

Para ilustrar o processo de mutação definido pela abordagem de matriz de permutação, considere os vetores:

$$\mathbf{x}_{r_1,g} = \begin{pmatrix} 2 \\ 4 \\ 1 \\ 5 \\ 3 \end{pmatrix}, \mathbf{x}_{r_2,g} = \begin{pmatrix} 1 \\ 4 \\ 3 \\ 5 \\ 2 \end{pmatrix}, \mathbf{x}_{r_3,g} = \begin{pmatrix} 1 \\ 3 \\ 4 \\ 5 \\ 2 \end{pmatrix}, \tag{2.17}$$

onde cada vetor é uma permutação válida. A matriz de permutação $\mathbf{P}$ que mapeia $\mathbf{x}_{r_3,g}$ em $\mathbf{x}_{r_2,g}$, sendo $\mathbf{P}\mathbf{x}_{r_3,g} = \mathbf{x}_{r_2,g}$, é dada por:

$$\mathbf{P} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \tag{2.18}$$

e, com isso, o novo vetor $\mathbf{v}_{i,g}$ obtido através da adaptação da mutação diferencial é dado por:

$$\mathbf{v}_{i,g} = \mathbf{P}\mathbf{x}_{r_1,g} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 4 \\ 1 \\ 5 \\ 3 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ 4 \\ 5 \\ 3 \end{pmatrix}. \tag{2.19}$$

O fator de ponderação ψ, similar à mutação diferencial original (2.12), pode ser usado para escalonar o efeito da matriz de permutação nas novas soluções (Price *et al.*, 2005). Para isso, o fator de ponderação $\psi \in [0,1]$ define a probabilidade de cada movimento ser realizado. Através da abordagem por matriz de permutação, todas as novas soluções obtidas serão sempre permutações válidas.

Para ilustrar o comportamento desta abordagem, considere a instância *kroA100* do TSP (Reinelt, 1995). A Figura 2.3 apresenta a evolução da população ao longo de 120 iterações do algoritmo. É facilmente observado que a abordagem é capaz de convergir para uma única solução, porém a solução para a qual o algoritmo converge é de baixa qualidade, não apresentando

nenhum padrão em termos de geometria.

2.3.2.2 Abordagem por matriz de adjacência

Em problemas onde as permutações definem ciclos, como no problema do caixeiro viajante, vetores como $(1,\ 2,\ 3,\ 4,\ 5)^{\mathrm{T}}$ e $(3,\ 4,\ 5,\ 1,\ 2)^{\mathrm{T}}$ representam a mesma solução. Estes vetores estão apenas rotacionados. No entanto, a abordagem por matriz de permutação não é capaz de identificar estas situações.

Devido à dificuldade em identificar vetores rotacionados através da abordagem por matriz de permutação, Price *et al.* (2005) apresentaram ainda uma outra abordagem, denominada abordagem por matriz de adjacência. Nela as soluções são representadas por matrizes de adjacência, e não por vetores de permutação de inteiros. A matriz de adjacência define o subgrafo do ciclo solução. Com esta estratégia, permutações iguais, mesmo que rotacionadas, geram a mesma matriz de adjacência. Assim, a diferença entre elas sempre será zero.

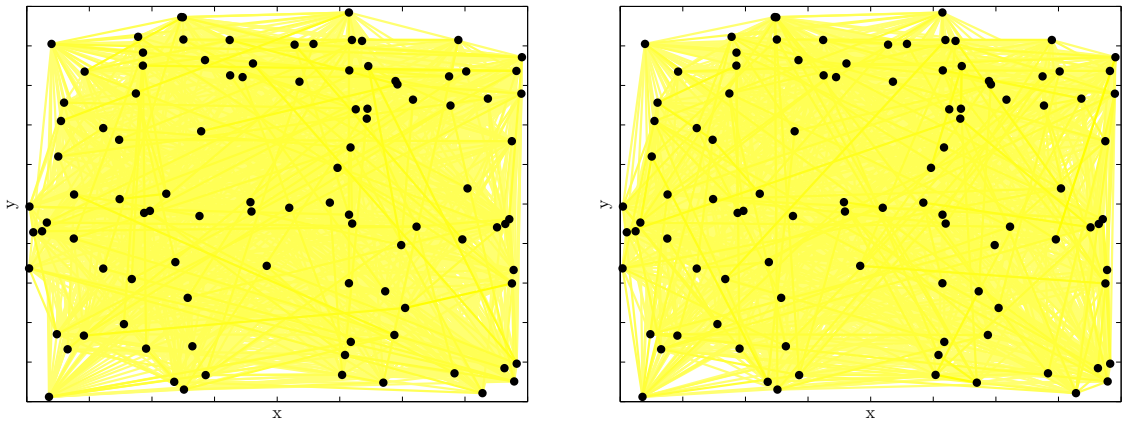
Considerando os seguintes vetores de permutação e suas respectivas matrizes de adjacência:

$$\mathbf{x}_{r_2,g} = \begin{pmatrix} 1 \\ 4 \\ 3 \\ 5 \\ 2 \end{pmatrix} \quad \equiv \quad \mathbf{X}_{r_2,g} = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}, \tag{2.20}$$

$$\mathbf{x}_{r_3,g} = \begin{pmatrix} 1 \\ 3 \\ 4 \\ 5 \\ 2 \end{pmatrix} \quad \equiv \quad \mathbf{X}_{r_3,g} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}, \tag{2.21}$$

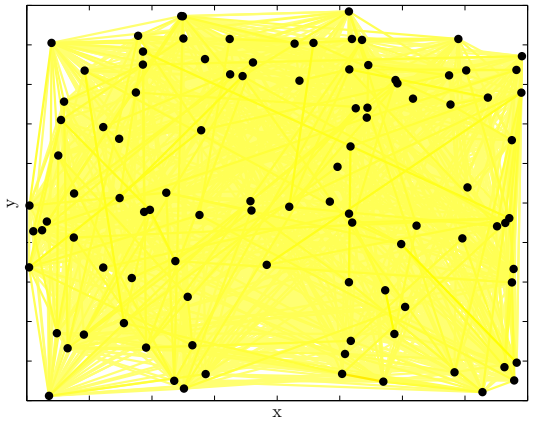
a diferença entre eles é definida por:

$$\mathbf{D}_{i,g} = \mathbf{X}_{r_2,g} \oplus \mathbf{X}_{r_3,g} = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}. \tag{2.22}$$

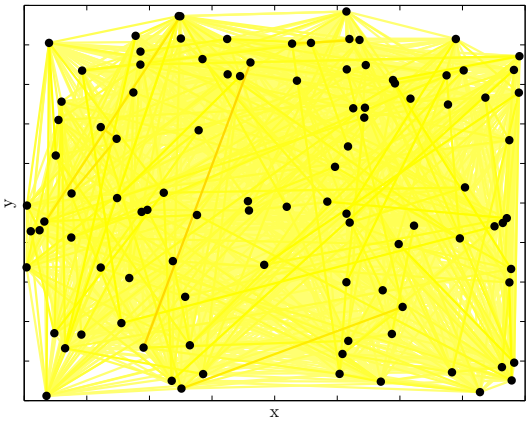


(a) População inicial

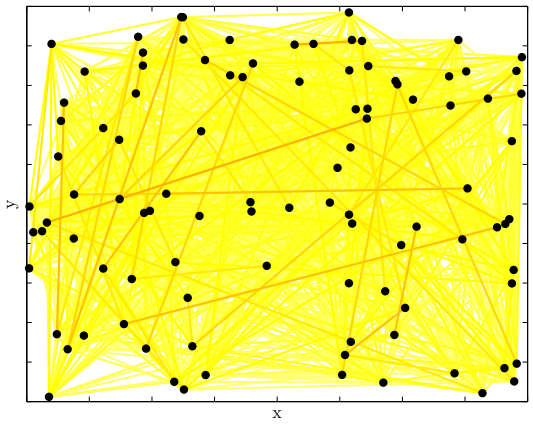
(b) 10ª iteração



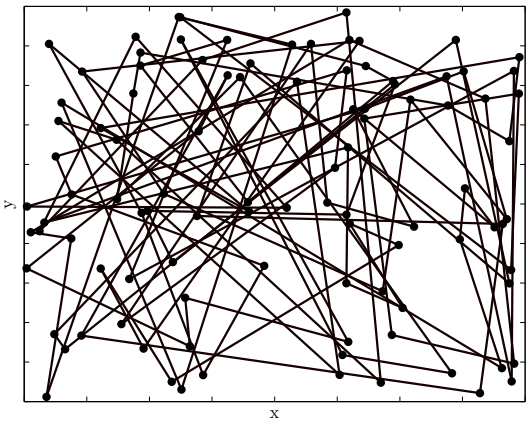
(c) 40ª iteração



(d) 70ª iteração



(e) 100ª iteração



(f) 120ª iteração

Figura 2.3 Comportamento da abordagem por matriz de permutação para uma instância de 100 cidades (círculos pretos) do TSP, ilustrado pela ocorrência de arestas na população ao longo das iterações. Quanto mais escura é a aresta, maior é o número de indivíduos na população em que ela aparece.

Após obter a diferença entre duas matrizes de adjacência, o resultado obtido é combinado à matriz de adjacência da solução base. No entanto, em problemas como o TSP, é improvável que a diferença ou a combinação da diferença com uma matriz de adjacência válida resulte em uma outra matriz de adjacência válida (Price *et al.*, 2005). Devido a isso, é necessário utilizar mecanismos de reparação para garantir que, no final do processo, o resultado seja uma matriz de adjacência válida.

2.3.2.3 Abordagem *relative position index*

Na abordagem *relative position index* (Lichtblau, 2009), as soluções são codificadas como vetores de permutação de inteiros, assim como é feito na abordagem por matriz de permutação. Porém os operadores de mutação e recombinação do DE original são mantidos inalterados.

Após a aplicação dos operadores de mutação e recombinação, os vetores gerados provavelmente possuirão valores fracionários. Assim, antes das soluções serem avaliadas pela função objetivo, elas são transformadas em vetores de permutação de inteiros válidos.

Para realizar a conversão dos vetores, a abordagem *relative position index* procede da seguinte maneira: dado o vetor $\mathbf{u}_{i,g}$, obtido após a mutação e recombinação, e considerando que uma solução válida é uma permutação de rótulos numéricos de 1 a n, o rótulo 1 é alocado na mesma posição onde se encontra o menor valor em $\mathbf{u}_{i,g}$, o rótulo 2 na posição do segundo menor valor em $\mathbf{u}_{i,g}$, e assim por diante. Para ilustrar esta estratégia, considere o vetor:

$$\mathbf{u}_{i,g} = (0,\ -2,\ 6.5,\ 3,\ 3.5)^{\mathrm{T}}\ . \tag{2.23}$$

O rótulo 1 é alocado na posição de menor valor, ou seja, na segunda posição do vetor. O rótulo 2 é alocado na primeira posição, onde se encontra do segundo menor valor. Seguindo esta estratégia, o rótulo 3 é alocado na quarta posição, o rótulo 4 na quinta posição e o rótulo 5 na terceira posição, resultando no seguinte vetor de permutação:

$$\mathbf{u}'_{i,g} = (2,\ 1,\ 5,\ 3,\ 4)^{\mathrm{T}}\ . \tag{2.24}$$

Com o uso dos operadores de mutação e recombinação do DE original, podem ocorrer situações onde o vetor $\mathbf{u}_{i,g}$ gerado possua dois ou mais valores repetidos. Nestes casos, estas soluções devem ser descartadas ou, então, utilizar algum critério de desempate.

A Figura 2.4 ilustra o comportamento da abordagem *relative position index* ao longo de 1000 iterações para a instância *kroA100* do TSP (Reinelt, 1995). Mesmo um grande número de iterações, o algoritmo não foi capaz de convergir.

2.3.2.4 Abordagem *smallest position value*

A abordagem *smallest position value* (Tasgetiren *et al.*, 2009) é semelhante à abordagem *relative position index*, divergindo apenas no processo de transformação dos vetores $\mathbf{u}_{i,g}$.

A abordagem *smallest position value* realiza a transformação em duas etapas. Primeiro, é atribuído um rótulo a cada $u^j_{i,g}$ e, em seguida, eles são ordenados.

Para atribuir os rótulos, é utilizada a mesma estratégia da abordagem *relative position index*, onde o rótulo 1 é atribuído ao menor valor, o rótulo 2 ao segundo menor valor, e assim por diante. Porém, na abordagem *smallest position value* é considerada apenas a parte inteira de $u^j_{i,g}$ para a atribuição dos rótulos. Após a atribuição dos rótulos, o vetor obtido é ordenado de acordo com a parte fracionária de $u^j_{i,g}$. Esta estratégia de atribuição de rótulos e ordenação após a atribuição é semelhante à estratégia utilizada definida pelo algoritmo *random keys* (Bean, 1994).

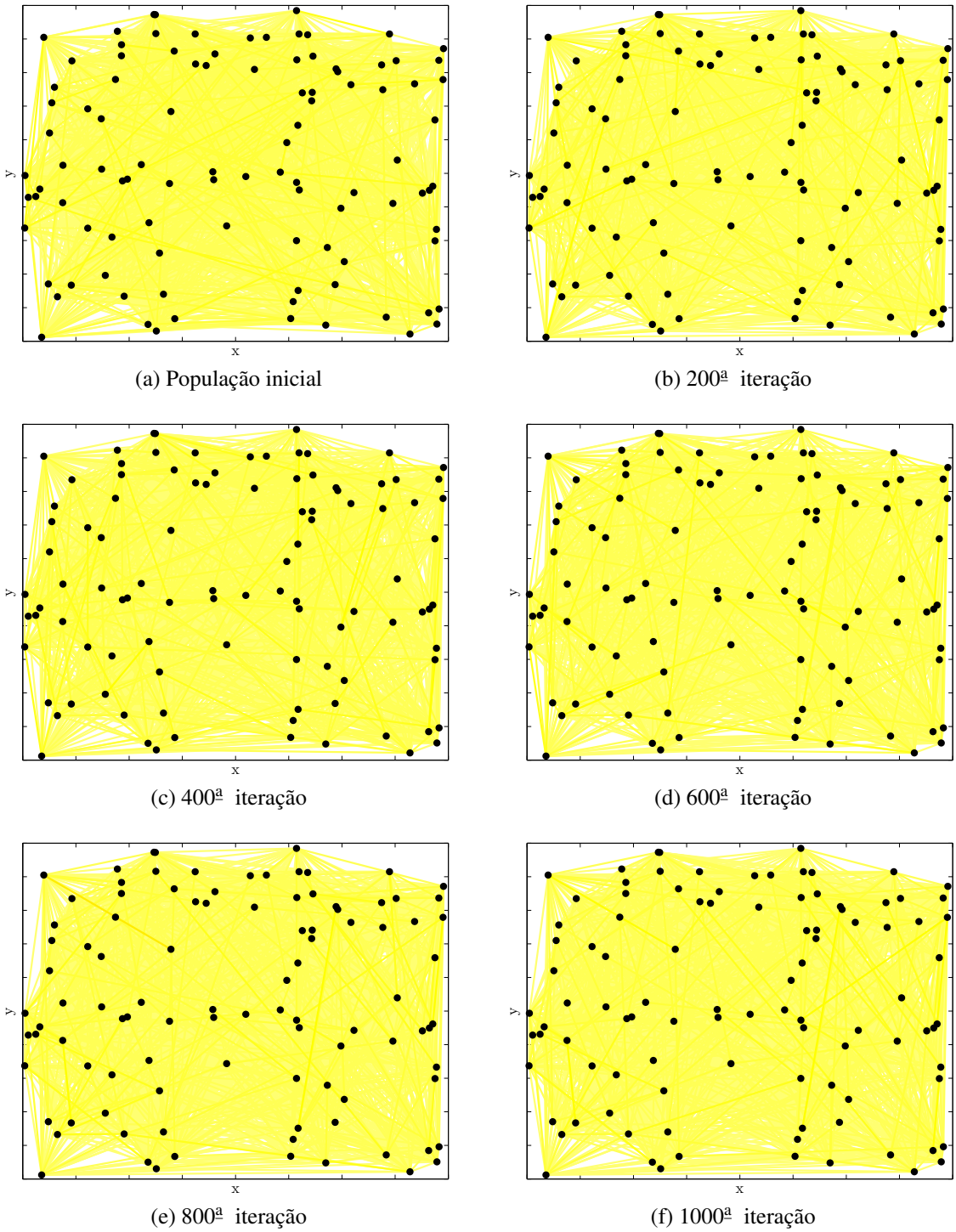
Para ilustrar esta abordagem, considere o vetor $\mathbf{u}_{i,g} = (3.4,\ 1.2,\ -4.6,\ 2.0,\ 7.1)^{\mathrm{T}}$. Após a atribuição dos rótulos, é obtido o vetor $\mathbf{u}'_{i,g} = (4,\ 2,\ 1,\ 3,\ 5)^{\mathrm{T}}$. Agora, ordenando este vetor de acordo com a parte fracionária dos valores em $\mathbf{u}_{i,g}$, é obtido o vetor de permutação de inteiros $\mathbf{u}''_{i,g} = (1,\ 3,\ 5,\ 2,\ 4)^{\mathrm{T}}$.

Assim como na abordagem *relative position index*, podem ocorrer vetores $\mathbf{u}_{i,g}$ com dois ou mais valores repetidos. Nestes casos, eles também devem ser descartados ou, então, utilizar algum critério de desempate.

2.3.2.5 Abordagem *forward/backward transformation*

A abordagem *forward/backward transformation*, apresentada por Onwubolu (2001), também utiliza os operadores de mutação e recombinação originais do DE, porém utilizando uma estratégia de mapeamento do espaço discreto para o contínuo e vice-versa.

Dessa forma, antes dos vetores serem submetidos aos operadores de mutação e recombinação eles são mapeados em vetores de valores reais. Esta etapa é denominada *forward*



(a) População inicial

(b) 200ª iteração

(c) 400ª iteração

(d) 600ª iteração

(e) 800ª iteração

(f) 1000ª iteração

Figura 2.4 Comportamento da abordagem *relative position index* para uma instância de 100 cidades (círculos pretos) do TSP, ilustrado pela ocorrência de arestas na população ao longo das iterações. Quanto mais escura é a aresta, maior é o número de indivíduos na população em que ela aparece.

transformation. Após o processo de mutação e recombinação, os vetores gerados são mapeados de volta em vetores de permutação de números inteiros, etapa denominada *backward transformation*.

Para a etapa *forward transformation* o algoritmo utiliza seguinte função de mapeamento:

$$\hat{\mathbf{x}}_{i,g} = -1 + \alpha \mathbf{x}_{i,g} \tag{2.25}$$

onde $\hat{\mathbf{x}}_{i,g}$ é o resultado do mapeamento do vetor de permutação de inteiros $\mathbf{x}_{i,g}$ em um vetor de números reais. O parâmetro α é um valor pequeno definido manualmente, em que os autores desta abordagem sugerem $\alpha = 500/999$.

Já a etapa *backward transformation*, mapeia o vetor $\hat{\mathbf{u}}_{i,g}$ obtido após as operações de mutação e recombinação para um vetor de números inteiros através da função:

$$u_{i,g}^{j} = \left\lfloor \frac{1 + \hat{u}_{i,g}^{j}}{\alpha} + 0.5 \right\rfloor . \tag{2.26}$$

Porém, a etapa *backward transformation* pode gerar vetores com valores inteiros repetidos, que representam soluções inválidas (Price *et al.*, 2005). Assim, é necessário adotar alguma estratégia para lidar com essa situação, seja através do uso de operadores de reparação ou simplesmente descartando tais soluções.

2.3.2.6 Abordagem por lista de movimentos

A abordagem por lista de movimentos (Prado *et al.*, 2010) define a diferença entre duas soluções candidatas como uma lista de movimentos que transformam um vetor de permutação de inteiros em outro. Esta ideia é semelhante à abordagem por matriz de permutação, no entanto, a abordagem por lista de movimentos define uma estratégia mais geral, uma vez que pode-se utilizar movimentos específicos para o problema.

Para ilustrar esta abordagem, considere os vetores $\mathbf{x}_{r_1,g} = (5,\ 3,\ 1,\ 4,\ 2)^{\mathrm{T}}$, $\mathbf{x}_{r_2,g} = (2,\ 1,\ 5,\ 4,\ 3)^{\mathrm{T}}$ e $\mathbf{x}_{r_3,g} = (1,\ 5,\ 4,\ 2,\ 3)^{\mathrm{T}}$. Primeiro é criada a lista de movimentos que leva $\mathbf{x}_{r_3,g}$ a $\mathbf{x}_{r_2,g}$. Neste exemplo cada movimento é a simples permutação entre pares de elementos de $\mathbf{x}_{r_3,g}$. Cada movimento é representado por uma tupla (i, j) indicando que o valor da posição i do vetor passará para a posição j e o valor da posição j passará para a posição i. Assim, a

lista de movimentos é dada por:

$$\mathcal{M} = \{(1,4),\ (2,4),\ (3,4)\}\ . \tag{2.27}$$

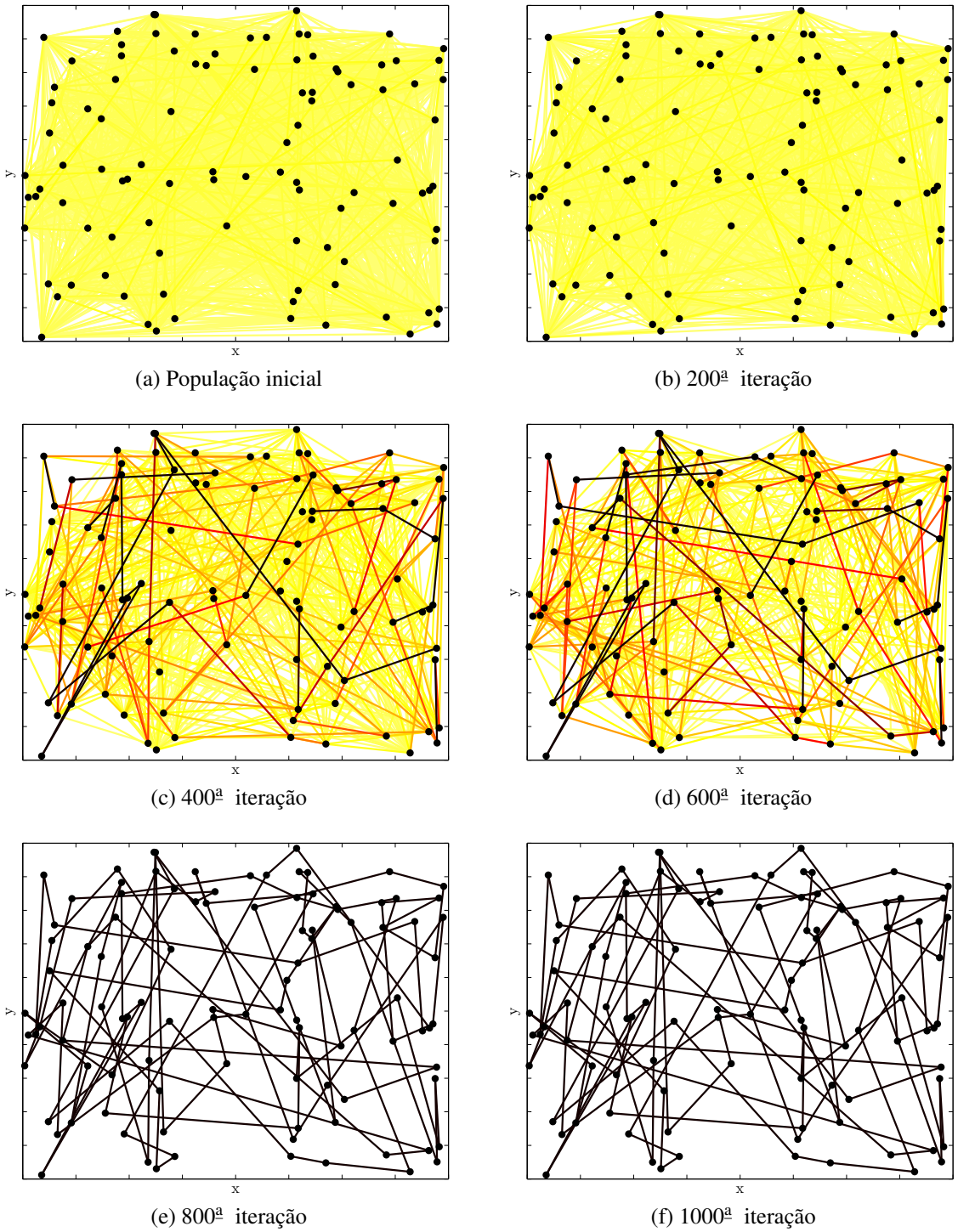
Após criada a lista de movimentos, eles são aplicados à solução base $\mathbf{x}_{r_1,g}$, gerando um novo vetor de permutação de inteiros:

$$\mathbf{v}_{i,g} = (4,\ 5,\ 3,\ 1,\ 2)^{\mathrm{T}}\ . \tag{2.28}$$

O fator de ponderação da mutação diferencial ψ é utilizado para controlar o tamanho da lista de movimentos. Definindo ψ como um valor no intervalo $[0,1]$, Prado *et al.* (2010) sugerem três estratégias:

- utilizar apenas os $\lceil \psi \times |\mathcal{M}| \rceil$ primeiros movimentos em $\mathcal{M}$ para modificar a solução base $\mathbf{x}_{r_1,g}$;
- percorrer sequencialmente a lista de movimentos $\mathcal{M}$, a partir do primeiro até o último movimento, onde cada movimento possui uma probabilidade ψ de ser utilizado para modificar a solução base $\mathbf{x}_{r_1,g}$;
- selecionar aleatoriamente $\lceil \psi \times |\mathcal{M}| \rceil$ movimentos de $\mathcal{M}$ para modificar a solução base $\mathbf{x}_{r_1,g}$.

A Figura 2.5 ilustra o comportamento da abordagem por lista de movimentos para a instância *kroA100* de 100 nós do TSP (Reinelt, 1995). Ao longo das iterações do algoritmo, as diferenças entre as soluções tendem a diminuir (Prado *et al.*, 2010), e assim, as listas de movimentos geradas são cada vez menores, indicando a capacidade do algoritmo de convergir para uma única região do espaço de busca, como acontece com o algoritmo DE original. No entanto, assim como ocorre com a abordagem por matriz de permutação, a solução para a qual o algoritmo converge é de baixa qualidade, não apresentando nenhum padrão em termos de geometria.



(a) População inicial

(b) 200ª iteração

(c) 400ª iteração

(d) 600ª iteração

(e) 800ª iteração

(f) 1000ª iteração

Figura 2.5 Comportamento da abordagem por lista de movimentos para uma instância de 100 cidades (círculos pretos) do TSP, ilustrado pela ocorrência de arestas na população ao longo das iterações. Quanto mais escura é a aresta, maior é o número de indivíduos na população em que ela aparece.

2.4 Conclusões

O uso de operadores de reparação, necessários nas abordagens por matriz de adjacência e *forward/backward transformation* fazem com que a informação herdada das soluções que a originaram sejam perdidas, já que estes operadores realizam modificações na solução sem levar em consideração estas informações.

Todas as adaptações do DE para otimização combinatória descritas neste capítulo, com exceção da abordagem por lista de movimentos, são aplicáveis apenas a problemas de otimização combinatória baseado em permutação. A abordagem por lista de movimentos define uma estratégia mais geral que pode ser também aplicada a problemas de otimização combinatória não baseados em permutação, simplesmente utilizando outros movimentos ao invés do movimento de troca.

A codificação utilizada por estas abordagens não permitem aos algoritmos manipularem as informações do problema, pois suas soluções são formadas por permutações de rótulos arbitrários e, ao realizar as operações matemáticas do DE, não existe nenhuma informação nos valores manipulados. Com isso, estas algoritmos não são capazes de aprender quais são os elementos mais promissores do problema para levar a melhores soluções. O real efeito destas abordagens não vai muito além de uma busca aleatória.

Como consequência disso, as abordagens *relative position index* e *smallest position value* apresentam dificuldades em convergir. Já as abordagens por matriz de permutação e lista de movimentos, apesar de conseguirem convergir, a geometria das soluções geradas não apresentam nenhum padrão.

Os bons resultados reportados nos trabalhos em que foram propostos são possivelmente devidos aos mecanismos de busca local e mecanismos de reparação utilizados em conjunto com estas abordagens (Storn, 2008).

Diante dos problemas apresentados pelas adaptações atuais do DE para otimização combinatória, no capítulo seguinte é introduzida uma codificação baseada em conjuntos para representação de solução e como ela pode ser utilizada juntamente com o DE, preservando o comportamento original deste algoritmo na solução desta classe de problemas.

CAPÍTULO 3

Algoritmo Proposto

Este capítulo introduz uma codificação baseada em conjuntos para representação de soluções de problemas de otimização combinatória em geral e seu uso com uma nova adaptação do algoritmo de evolução diferencial para otimização combinatória, sejam baseados em permutação ou não. Além de apresentar sua estrutura geral e as diferenças em relação ao algoritmo original, o capítulo discute também seu comportamento ao longo do processo de otimização.

3.1 Codificação baseada em conjuntos

Como discutido no capítulo anterior, problemas combinatórios, sejam baseados em permutação ou não, podem ser vistos como um problema onde deve-se definir um subconjunto a partir do conjunto de possíveis elementos para construção de uma solução. Desta forma, conjuntos fornecem uma representação geral para problemas de natureza combinatória

Quando um problema de otimização combinatória é modelado como um problema de programação inteira binária, os componentes do problema para os quais deve-se escolher entre fazer parte da solução ou não, são modelados através de variáveis binárias. Geralmente são esses os componentes que possuem as informações mais relevantes do problema, por exemplo, os itens para o problema da mochila 0-1 e as arestas para o problema do caixeiro viajante.

Através de uma codificação de soluções baseada em conjuntos, os algoritmos podem manipular os componentes que realmente possuem informações relevantes do problema a ser resolvido, evitando assim codificações como as que utilizam rótulos arbitrários, que são possivelmente a causa pelo qual as atuais adaptações do DE não são capazes de reter as características de convergência do DE básico.

Ao combinar a codificação baseada em conjuntos com a estrutura do DE, os operadores aritméticos utilizados na etapa da mutação diferencial não podem ser aplicados de maneira direta, pois tais operadores trabalham em cima de vetores. Assim, tais operadores necessitam ser

substituídos por operações sobre conjuntos que retenham a essência dos operadores originais. Tais adaptações são descritas na seção seguinte.

3.2 Adaptação do DE para otimização combinatória

Nesta seção é introduzido um algoritmo baseado no DE para solucionar problemas de natureza combinatória. Diferente do algoritmo original e das adaptações existentes para otimização combinatória, as soluções são representadas por conjuntos, como descrito na seção anterior, e não mais por vetores.

Devido à alteração na representação das soluções, os operadores também necessitam de modificações. Ao invés de utilizar as operações aritméticas sobre vetores, como descritas em (3.1), são adotados operações sobre conjuntos. Assim, a mutação é realizada como:

$$\mathbb{V}_{i,g} = \mathbb{X}_r \cup \psi \cdot (\mathbb{X}_{r_1,g} \oplus \mathbb{X}_{r_2,g}) \tag{3.1}$$

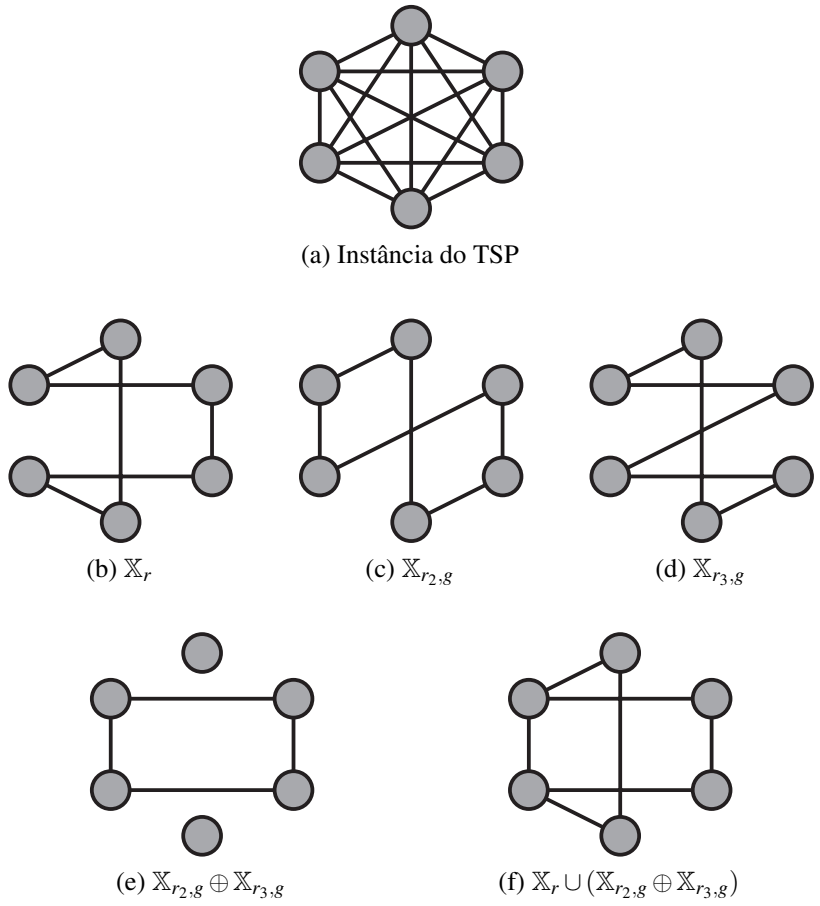
onde $\mathbb{X}_r$ é uma solução candidata gerada aleatoriamente e $r_1 \neq r_2 \neq i \in [1, n_p]$ são índices gerados aleatoriamente. As operações de soma e subtração são substituídas pelas operações de união e ou-exclusivo, respectivamente. Como as soluções são codificadas como conjuntos, estas operações podem ser aplicadas naturalmente. A Figura 3.1 ilustra a aplicação do operador de mutação em uma instância do TSP.

O operador lógico ou-exclusivo desempenha o papel da operação de subtração existente na mutação, uma vez que seu resultado é um subconjunto composto apenas por aqueles elementos diferentes entre os dois conjuntos, podendo ser escrito por:

$$\mathbb{X}_{r_1,g} \oplus \mathbb{X}_{r_2,g} = \{\mathbb{X}_{r_1,g} \cup \mathbb{X}_{r_2,g}\} \setminus \{\mathbb{X}_{r_1,g} \cap \mathbb{X}_{r_2,g}\}. \tag{3.2}$$

O fator de ponderação da mutação $\psi \in [0, 1]$ é utilizado para controlar o tamanho do conjunto resultante $\mathbb{V}_{i,g}$, assim como é realizado na abordagem por lista de movimentos. Dessa forma, pode-se utilizar $\lceil \psi | \mathbb{V}_{i,g} | \rceil$ elementos em $\mathbb{V}_{i,g}$ aleatoriamente escolhidos ou selecionar cada elemento com probabilidade ψ. A estratégia de selecionar os $\lceil \psi | \mathbb{V}_{i,g} | \rceil$ primeiros elementos não é recomendada pois esta estratégia poderia ser tendenciosa à implementação, já que um conjunto não define ordenação entre seus elementos.

Diferente do que ocorre na operação de mutação do DE clássico, onde as três soluções

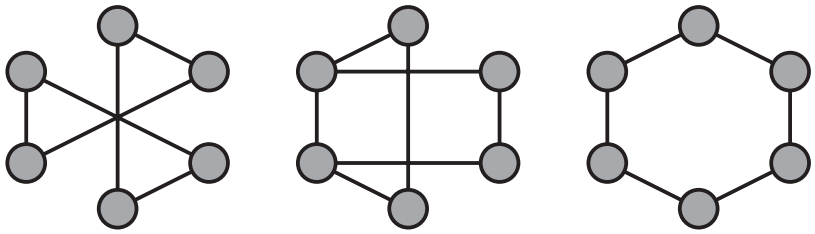


(a) Instância do TSP

(b) $\mathbb{X}_r$ (c) $\mathbb{X}_{r_2,g}$ (d) $\mathbb{X}_{r_3,g}$

(e) $\mathbb{X}_{r_2,g} \oplus \mathbb{X}_{r_3,g}$ (f) $\mathbb{X}_r \cup (\mathbb{X}_{r_2,g} \oplus \mathbb{X}_{r_3,g})$

Figura 3.1 Estratégia de mutação proposta aplicada a uma instância do TSP: (a) O grafo completo da instância do TSP; (b) Solução criada aleatoriamente; (c) e (d) Soluções candidatas aleatoriamente escolhida da população. O resultado do operador lógico ou-exclusivo entre (c) e (d) é apresentado em (e), e a solução mutante, construída a partir da união entre (b) e (e), é apresentada em (f).



(a) $\mathbb{X}_{i,g}$ (b) $\mathbb{V}_{i,g}$ (c) $\mathbb{U}_{i,g}$

Figura 3.2 Recombinação através da solução de um subproblema aplicado em uma instância do TSP: (a) Solução candidata *target*; (b) Solução candidata mutante. O algoritmo deve encontrar uma solução utilizando apenas os elementos que estão presentes em (a) e (b), resultando na solução *trial* (c).

envolvidas no processo de mutação são aleatoriamente escolhidas da população, a nova abordagem obtém apenas duas, sendo a solução base $\mathbb{X}_r$ criada aleatoriamente. Isso é necessário já que, para problemas muito grandes, podem existir elementos que não estejam presentes em nenhuma das soluções da população e assim, soluções formadas por esses elementos faltantes nunca seriam exploradas.

Após a mutação, a recombinação é realizada através da criação de uma solução *trial* $\mathbb{U}_{i,g}$ utilizando apenas elementos presentes na solução *target* $\mathbb{X}_{i,g}$ e no conjunto $\mathbb{V}_{i,g}$ gerado no processo de mutação. A escolha dos elementos que irão compor $\mathbb{U}_{i,g}$ pode ser vista como a solução de um subproblema da forma:

$$\text{Encontrar } \mathbb{U}_{i,g} \in \mathcal{F} \ : \ \mathbb{U}_{i,g} \subseteq \{\mathbb{X}_{i,g} \cup \mathbb{V}_{i,g}\}. \tag{3.3}$$

Este subproblema define um subconjunto limitado do espaço de busca original. Como este subconjunto inclui a solução *target* $\mathbb{X}_{i,g}$ que é uma solução válida, o subproblema sempre será factível. A Figura 3.2 ilustra o operador de recombinação para o TSP, considerando o mesmo esquema do exemplo do operador de mutação (Figura 3.1), que equivale ao cruzamento geométrico (Moraglio e Poli, 2004, 2011).

Para resolver o subproblema gerado, métodos de busca local podem ser empregados a partir da solução *target* $\mathbb{X}_{i,g}$, limitando-se apenas à avaliação de movimentos válidos para o subproblema, reduzindo assim o número de movimentos testados a cada iteração da busca local.

Como o subproblema resultante é menor que o problema original, em alguns casos pode ser possível utilizar métodos exatos para encontrar uma nova solução. Além disso, a solução *target* $\mathbb{X}_{i,g}$ pode ser definida como solução incumbente. Em métodos como o *branch-and-*

bound, o conhecimento prévio de uma boa solução pode reduzir significativamente o tempo necessário para se resolver um problema, pois tende a gerar mais podas na árvore de busca (Wolsey, 1998). No entanto, os subproblemas resolvidos possuem a mesma complexidade do problema original, apenas a sua dimensão é reduzida. Ou seja, se um problema é classificado como $\mathcal{NP}$-difícil, ele continuará sendo $\mathcal{NP}$-difícil.

Em alguns problemas, mesmo reduzindo a sua dimensão, métodos exatos ainda podem ser inviáveis. Exemplo disso são os problemas de programação inteira que possuem uma função objetivo não-linear ou restrições não-lineares, que violam as suposições geralmente necessárias por métodos exatos para fornecer garantias de convergência (Belotti *et al.*, 2013).

A ideia principal do algoritmo proposto é utilizar a estrutura do DE para definir subproblemas que sejam menores e, possivelmente, mais fáceis de resolver se comparados ao problema original. Por exemplo, considere o TSP onde o número de elementos que compõem o problema são as arestas de um grafo completo. Em um grafo completo com n nós exitem $n(n-1)/2$ arestas. No pior caso, que ocorre quando as quatro soluções candidatas envolvidas no processo de mutação e recombinação não possuem nenhuma aresta em comum, o subproblema a ser resolvido na etapa de recombinação terá $4n$ arestas. Isso significa que enquanto o problema original cresce de forma quadrática com o número de nós, os subproblemas crescem de forma linear.

Os aspectos restantes do DE original, como operador de seleção e critério de parada são mantidos inalterados. Assim, ao final de sua execução, a melhor solução encontrada é retornada. O Algoritmo 3.1 apresenta a estrutura geral do algoritmo proposto.

3.3 Comportamento do algoritmo

O algoritmo proposto possui a capacidade de auto-adaptação, similar ao DE original (Storn e Price, 1997). Nas iterações iniciais, o algoritmo desempenha um papel de exploração. O operador ou-exclusivo tende a gerar conjuntos com grande número de elementos, já que a população é mais diversificada. Assim, os subproblemas gerados nas iterações iniciais são maiores e o algoritmo explora regiões maiores do espaço de busca.

À medida em que a população começa a convergir, alguns elementos tendem a desaparecer enquanto aqueles que compõem as melhores soluções tendem a permanecer presentes na população nas iterações seguintes. Assim, o operador ou-exclusivo passa a gerar conjun-

Entrada: $f(\cdot) : \mathcal{F} \mapsto \mathbb{R}$: função objetivo
Entrada: n_p: tamanho da população
Entrada: ψ: fator de escalonamento da mutação

// Gera a população inicial
Inicializar soluções $\mathbb{X}_{1,0}, \mathbb{X}_{2,0}, \ldots, \mathbb{X}_{n_p,0} \in \mathcal{F}$;
$g \leftarrow 0$;
repita
  para $i \leftarrow 1$ **até** n_p **faça**
    // Mutação
    $\mathbb{V}_{i,g} \leftarrow \mathbb{X}_r \cup \psi \cdot (\mathbb{X}_{r_1,g} \oplus \mathbb{X}_{r_2,g})$; // $r_1 \neq r_2 \neq i \in [1, n_p]$
    // Recombinação
    encontrar $\mathbb{U}_{i,g} \in \mathcal{F} : \mathbb{U}_{i,g} \subseteq \{\mathbb{V}_{i,g} \cup \mathbb{X}_{i,g}\}$
    // Seleção
    se $f(\mathbb{U}_{i,g}) \leq f(\mathbb{X}_{i,g})$ **então**
      $\mathbb{X}_{i,g+1} \leftarrow \mathbb{U}_{i,g}$;
    senão
      $\mathbb{X}_{i,g+1} \leftarrow \mathbb{X}_{i,g}$;
    fim
  fim
  $g \leftarrow g + 1$;
até *critério de parada ser satisfeito*;
retorna *Melhor solução* $\mathbb{X}_{i,g}$ *encontrada*

Algoritmo 3.1: Algoritmo proposto

tos com menos elementos, o que faz com que o algoritmo passe a desempenhar um papel de intensificação, explorando apenas regiões menores do espaço de soluções.

Ao longo do processo iterativo, o algoritmo proposto é capaz de identificar os elementos mais promissores e construir subproblemas que levam a soluções melhores. Para ilustrar este comportamento, considere a instância *kroA100* do TSP com 100 cidades (Reinelt, 1995). A Figura 3.3 apresenta a evolução da população ao longo de 50 iterações. Na Figura 3.3a estão as arestas presentes nas soluções da população inicial, gerada aleatoriamente. Pode ser facilmente observado o grande número de diferentes arestas, cobrindo a maior parte do grafo.

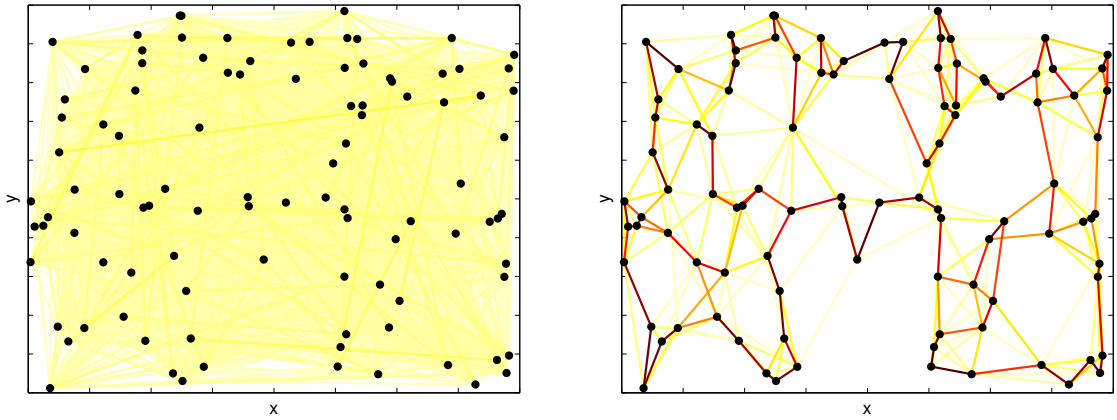
As Figuras 3.3a-3.3f ilustram a evolução da população, mostrando as arestas presentes nas soluções em iterações específicas. Pode ser observado que apenas um subconjunto reduzido de arestas continuam presentes nas últimas iterações. Na Figura 3.3f, pode-se observar que as arestas mais frequentes na população são aquelas que formam a melhor solução encontrada e, apenas um pequeno número de arestas que não fazem parte da melhor solução continuam presentes em outras soluções da população.

A população tende a convergir para uma única solução, assim como o algoritmo DE original para variáveis contínuas. No entanto, é necessário investigar a qualidade das soluções para a qual o algoritmo converge. No capítulo seguinte são realizados experimentos computacionais para investigar o desempenho e comportamento do algoritmo proposto em alguns problemas de otimização combinatória.

3.4 Conclusões

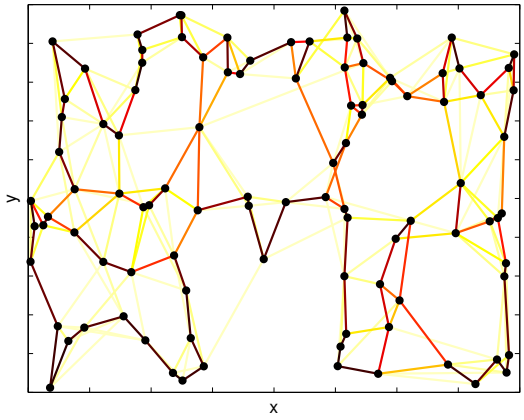
Neste capítulo é introduzida uma abordagem alternativa para uso do algoritmo de evolução diferencial para otimização combinatória. Nesta abordagem, as soluções são codificadas como conjuntos, sendo assim uma representação geral para problemas desta classe, sejam baseados em permutação ou não. Além disso, tal codificação possibilita ao algoritmo manipular os elementos utilizados na construção de soluções de maneira mais inteligente, evitando a busca aleatória das outras adaptações existentes do DE para otimização combinatória.

Os operadores definidos para esta nova abordagem possibilitam que o DE seja utilizado para definir subproblemas menores e, possivelmente, mais fáceis de serem resolvidos. Em alguns casos o subproblema é pequeno o suficiente para que métodos exatos possam ser utilizados como estratégia de busca local.

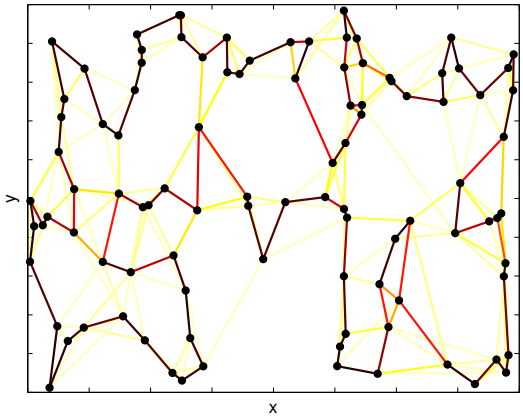


(a) População inicial

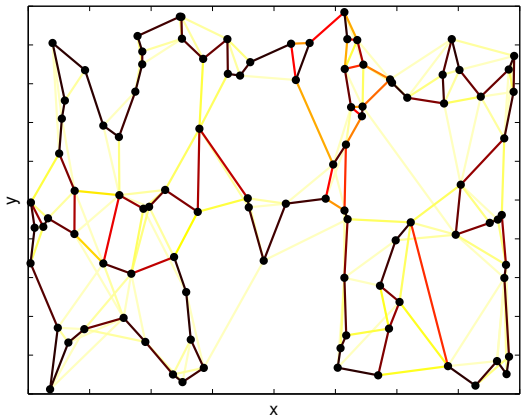
(b) 10ª iteração



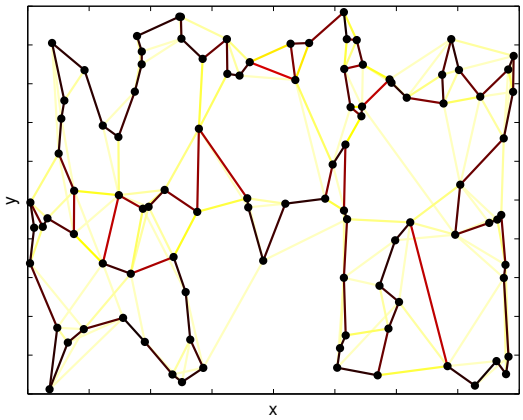
(c) 20ª iteração



(d) 30ª iteração



(e) 40ª iteração



(f) 50ª iteração

Figura 3.3 Comportamento do algoritmo proposto para uma instância de 100 cidades (círculos pretos) do TSP, ilustrado pela ocorrência de arestas na população ao longo das iterações. Quanto mais escura é a aresta, maior é o número de indivíduos na população em que ela aparece.

Diferente das abordagens existentes até então, a abordagem proposta para resolver problemas de otimização combinatória através do DE preserva as características que atraíram as atenções dos pesquisadores. Esta nova abordagem é capaz de convergir para uma única solução do espaço de busca selecionando os componentes que são mais promissores a compor soluções cada vez melhores.

CAPÍTULO 4

Experimentos Computacionais e Resultados

Neste capítulo são apresentados e discutidos os resultados referentes ao desempenho da adaptação do algoritmo de evolução diferencial para otimização combinatória, juntamente com a codificação de soluções por conjuntos, apresentados neste trabalho. Primeiro é avaliado o desempenho do algoritmo proposto em relação às outras adaptações do DE já existentes. Em seguida, é avaliado o desempenho do algoritmo proposto na solução de um problema de otimização combinatória envolvendo não-linearidades, onde métodos exatos tendem a apresentar dificuldades de convergência até mesmo para instâncias pequenas. Os resultados obtidos são comparados aos obtidos por outras abordagens: método exato e heurísticas de buscas local.

4.1 Problemas teste

Foram escolhidos dois problemas de otimização combinatória para avaliar o desempenho da adaptação do algoritmo proposto. Estes problemas são o problema do caixeiro viajante e o problema de agrupamento centrado capacitado.

A escolha destes problemas específicos se deve às suas características. Primeiro, deseja-se avaliar o desempenho do algoritmo proposto em relação às outras adaptações já existentes do DE. Para isso, foi escolhido o algoritmo do caixeiro viajante, já que as adaptações existentes basicamente se limitam a problemas de otimização combinatória baseados em permutação, e o problema do caixeiro viajante é um exemplo clássico deste tipo de problema. Além disso, é conhecido o ótimo para diversas instâncias deste problema, assim, além de comparar o desempenho do algoritmo proposto com as outras abordagens, é possível avaliar também a qualidade das soluções retornadas em relação ao ótimo global.

O problema do caixeiro viajante já é um problema de otimização combinatória bastante explorado, existindo métodos exatos capazes de resolver instâncias de grande porte em tempo

Tabela 4.1 Instâncias do TSP

Instância	*n*	Ótimo	Instância	*n*	Ótimo	Instância	*n*	Ótimo
att48	48	10628	kroC100	100	20749	pr144	144	58537
berlin52	52	7542	kroD100	100	21294	kroA150	150	26524
st70	70	675	kroE100	100	22068	kroB150	150	26130
pr76	76	108159	lin105	105	14379	pr152	152	73682
rat99	99	1211	pr107	107	44303	rat195	195	2323
kroA100	100	21282	pr124	124	59030	kroA200	200	29368
kroB100	100	22141	pr136	136	96772	kroB200	200	29437

aceitável[1]. Assim, para avaliar o desempenho do algoritmo proposto em problemas ainda mais complexos, é escolhido o problema de agrupamento centrado capacitado.

O problema de agrupamento centrado capacitado possui a função objetivo não-linear, o que tende a dificultar a convergência dos métodos exatos tradicionais até mesmo para instâncias de pequeno porte.

É importante ressaltar que o objetivo deste trabalho não é propor algoritmos específicos para estes problemas em particular, mas sim um algoritmo para otimização de problemas combinatórios de maneira geral, sejam baseados em permutação ou não.

As subseções seguintes descrevem em mais detalhes os problemas de otimização escolhidos e suas instâncias, utilizadas nos experimentos. Além disso, são descritas as estratégias adotadas para resolver os subproblemas gerados ao longo da execução do algoritmo proposto.

4.1.1 O problema do caixeiro viajante

Para o problema do caixeiro viajante (TSP, do inglês *traveling salesman problem*), introduzido no Capítulo 2, são utilizadas 21 instâncias da base de dados do TSPLIB (Reinelt, 1995) com número de nós variando entre 48 e 200. Para todas estas instâncias a solução ótima é conhecida. Na Tabela 4.1 são apresentados o número de nós (coluna *n*) e o valor ótimo para a função objetivo (Reinelt, 1995) das 21 instâncias utilizadas.

[1] Apesar de existir métodos capazes de resolver instâncias de grande porte para o caixeiro viajante de maneira exata, nenhum destes métodos possui complexidade de tempo polinomial.

4.1.1.1 Definição e solução dos subproblemas

Na etapa de recombinação, os subproblemas são definidos pelo conjunto de arestas obtidos pela união do conjunto resultantes da etapa de mutação com as arestas da solução *target*, conforme ilustrado anteriormente na Figura 3.2.

O subproblema é resolvido através do *solver* Gurobi[2], limitando seu tempo de execução a 10 segundos para cada subproblema resolvido. Os subproblemas são escritos através do modelo matemático definido por (2.7)–(2.11), utilizando a estratégia *lazy constraints* para as restrições de eliminação de subciclos (2.10). Através da estratégia *lazy constraints* as restrições de eliminação de subciclos são inseridas à medida em que se tornam necessárias até que uma solução válida seja encontrada, já que o número destas restrições cresce exponencialmente em função do número de nós.

Como a solução *target* é uma solução válida para cada sub-problema ao qual ela está associada, esta mesma solução é definida como ponto inicial para o Gurobi. Assim, no pior caso o Gurobi retorna a própria solução *target* ao final dos 10 segundos, caso não tenha encontrado nenhuma outra solução válida de melhor qualidade.

4.1.2 O problema de agrupamento centrado capacitado

O problema de agrupamento centrado capacitado (CCCP, do inglês *capacitated centered clustering problem*) (Negreiros e Palhano, 2006), consiste em definir p grupos de capacidade limitada com menor dissimilaridade em uma rede com n pontos. Cada ponto $i = \{1,\ldots,n\}$ possui uma demanda d_i e os grupos $j = \{1,\ldots,p\}$ possuem capacidade limitada igual a Q_j. A

[2]O Gurobi é um *solver* comercial para solução de problemas de programação matemática. Para otimização de problemas de programação inteira e programação inteira mista, o Gurobi utiliza o algoritmo *branch-and-bound* em conjunto com técnicas de plano de corte e heurísticas (Gurobi Optimization Inc., 2014).

formulação original do CCCP, definida por Negreiros e Palhano (2006), é dada por (4.1)–(4.6):

$$\min \sum_{i=1}^{n} \sum_{j=1}^{p} \| \vec{a}_i - \vec{q}_j \|^2 x_{ij} \tag{4.1}$$

$$\sum_{j=1}^{p} x_{ij} = 1 \qquad \forall\, i = 1, \ldots, n \tag{4.2}$$

$$\sum_{i=1}^{n} x_{ij} = n_j \qquad \forall\, j = 1, \ldots, p \tag{4.3}$$

$$\sum_{i=1}^{n} \vec{a}_i x_{ij} \leq n_j \vec{q}_j \qquad \forall\, j = 1, \ldots, p \tag{4.4}$$

$$\sum_{i=1}^{n} d_i x_{ij} \leq Q_j \tag{4.5}$$

$$\vec{a}_i, \vec{q}_j \in \mathbb{R}^{\ell}, \;\; n_j \in \mathbb{N}, \;\; x_{ij} \in \{0,1\} \tag{4.6}$$

onde:

- x_{ij} é uma variável binária que indica se o nó i está alocado ao agrupamento j ($x_{ij} = 1$) ou não ($x_{ij} = 0$);
- $\vec{q}_j$ representa a posição do centroide do agrupamento j;
- n_j representa o número de nós alocados ao agrupamento j;
- $\vec{a}_i$ representa a posição do nó i.

O CCCP é um problema $\mathcal{NP}$-difícil e, além disso, o cálculo de dissimilaridade é dado por uma função não-linear, dificultando ainda mais sua solução por métodos exatos.

Apesar da formulação (4.1)–(4.6) ser de fácil entendimento, ela não pode ser resolvida diretamente através de *solvers* como o Gurobi. Para que isso seja possível, Stefanello e Müller (2009) apresentaram uma formulação alternativa para o CCCP. Considerando a mesma notação do modelo acima, M como sendo um valor suficientemente grande e $\vec{c}_{ij} \in \mathbb{R}^{\ell}$ como um vetor que definem o valor absoluto da diferença das coordenadas entre o nó i e o agrupamento j, onde c_{ijk} representa o valor absoluto da diferença da k-ésima coordenada, o CCCP pode ser

Tabela 4.2 Instâncias do CCCP

Instância	n	p	Q	Melhor
SJC1	100	10	720	17359,75
SJC2	200	15	840	33181,65
SJC3a	300	25	740	45358,23
SJC3b	300	30	740	40661,94
SJC4a	402	30	840	61931,60
SJC4b	402	40	840	52214,55
Doni1	1000	6	200	3021,41
Doni2	2000	6	400	6080,70
Doni3	3000	8	400	8438,96
Doni4	4000	10	400	10854,48
Doni5	5000	12	450	11134,94
Doni6	10000	23	450	15722,67
Doni7	13221	30	450	18596,74

modelado por (4.7)–(4.11):

$$\min \sum_{i=1}^{n}\sum_{j=1}^{p}\sum_{k=1}^{\ell} c_{ijk}^2 \tag{4.7}$$

$$\sum_{j=1}^{p} x_{ij} = 1 \qquad \forall\, i = 1,\ldots,n \tag{4.8}$$

$$\sum_{i=1}^{n} d_i x_{ij} \leq Q_j \tag{4.9}$$

$$\mid \vec{q}_j - \vec{a}_i \mid \leq \vec{c}_{ij} + (1 - x_{ij})M \qquad \forall\, i = 1,\ldots,n,\ \ \forall\, j = 1,\ldots,p \tag{4.10}$$

$$c_{ijk} \geq 0,\ \ x_{ij} \in \{0,1\} \tag{4.11}$$

Foram utilizadas treze instâncias do CCCP com número de nós variando entre 100 e 13221 (Lorena, 2013). A Tabela 4.2 apresenta algumas informações a respeito das instâncias utilizadas: o número de nós, o número máximo de agrupamentos, a capacidade máxima dos agrupamentos e os melhores resultados conhecidos para as instâncias[3]. Apesar do problema permitir capacidades diferentes entre os agrupamentos, todas as instâncias utilizadas definem agrupamentos de capacidade homogênea.

[3] Valores reportados no trabalho de Chaves e Lorena (2011).

O conjunto de instâncias *SJC* possui nós com demandas heterogêneas, ao contrário das instâncias da conjunto *Doni*, onde as demandas são homogêneas e iguais 1.

4.1.2.1 Definição e solução dos subproblemas

O CCCP pode ser modelado como um grafo bipartido onde um grupo de vértices é formado pelos nós $i = \{1,\ldots,n\}$ e o outro grupo de vértices é formado pelos agrupamentos $j = \{1,\ldots,p\}$. Assim, uma solução para este problema pode ser representada por um conjunto de arestas (i, j) que indicam a alocação do nó i ao agrupamento j. Dessa forma, estas arestas definirão o conjunto de componentes do problema que serão manipulados pelo algoritmo proposto neste trabalho.

Em testes preliminares utilizando o Gurobi para resolver os subproblemas, era necessário muito tempo até que uma solução de melhor qualidade que a solução *target* (fixada como ponto de partida) pudesse ser encontrada. Devido a isso foi adotada uma estratégia heurística que, a partir da solução *target* realiza uma busca local, limitado pelo subproblema, realizando movimentos de troca. Este movimento realoca um nó i em um agrupamento j' para um outro agrupamento j''. Apenas os movimentos válidos para o subproblema são testados, reduzindo assim o número de movimentos avaliados em uma busca local.

A busca local teve o tempo de execução limitado a 3 segundos para cada subproblema resolvido. Para avaliar os movimentos, é criada uma lista com de todos os movimentos válidos para o subproblema e então, a busca local avalia estes movimentos em ordem aleatória. Cada movimento é avaliado uma única vez a cada iteração da busca local. Esta estratégia evita que a busca local seja tendenciosa a uma ordenação específica dos movimentos caso o limite de tempo para a busca local seja alcançado antes que todos movimentos tenham sido avaliados.

4.2 Planejamento estatístico dos experimentos

Todos os algoritmos testados no experimento foram implementados em Java (JDK 7) e os experimentos realizados em um computador com 2 processadores Intel®Xeon®E5-2640, 94 GB de memória RAM, sistema operacional GNU/Linux Ubuntu Server 12.04.3 de 64 bits. Nos testes em que o Gurobi foi utilizado, seus parâmetros foram mantidos com valores padrão, apenas limitando o uso de uma única *thread* de processamento e o tempo máximo de execução.

Em todos os experimentos para comparação de desempenho entre algoritmos, modelos estatísticos foram utilizados para a realização de testes de significância. O número inicial de réplicas para cada problema foi de 5 execuções dos algoritmos em cada instância, mas a maior variância dos resultados no CCCP gerou a necessidade de um maior tamanho amostral para a detecção de efeitos de interesse. Nesse problema, uma amostra de tamanho 30 foi utilizada.

Para os testes de significância um modelo de análise de variância com blocos generalizados (Addelman, 1969; Montgomery, 2008) foi utilizado, sendo *Algoritmo* o fator de interesse e *Problema* o fator de bloqueamento. Nesse modelo a variabilidade total dos dados é particionada entre o efeito principal dos algoritmos, dos problemas, e da interação problema-algoritmo. O modelo estatístico correspondente é dado por:

$$y_{ijk} = \mu_0 + \alpha_i + \beta_j + \alpha\beta_{ij} + \varepsilon_{ijk} \tag{4.12}$$

onde y_{ijk} é o valor da métrica de desempenho obtido pelo i-ésimo algoritmo na j-ésima instância da k-ésima replicação. As componentes μ_0 é a média global, α_i é a variabilidade devida ao i-ésimo algoritmo, β_j é a variabilidade devida à j-ésima instância, $\alpha\beta_{ij}$ é a variabilidade devida à interação entre o i-ésimo algoritmo com a j-ésima instância. O termo residual ε_{ijk} representa a variabilidade devida aos outros fatores não modelados neste experimento, por exemplo, o efeito da inicialização aleatória dos algoritmos.

A ANOVA[4] é aplicada para testar as hipóteses nulas de ausência de diferença no desempenho dos algoritmos contra a hipótese alternativa de que pelo menos um algoritmo apresenta diferença em relação a um outro algoritmo:

$$\begin{cases} H_0 \ : \ \alpha_i = 0, & \forall i \\ H_1 \ : \ \alpha_i \neq 0, & \text{para qualquer } i \end{cases} . \tag{4.13}$$

As premissas da ANOVA foram verificadas utilizando análise gráfica dos resíduos e testes formais (Crawley, 2007). Nos casos em que violações graves foram encontradas, foi utilizada uma transformação em *ranks* para estabilizar os resíduos (Montgomery, 2008).

Após os testes estatísticos de significância, a técnica de *bootstrap* (Davison e Hinkley, 1997) foi utilizada para o cálculo de estimativas pontuais e intervalos de confiança para o desempenho médio geral (após a remoção do efeito dado pelas instâncias) e o desempenho médio dos algoritmos para cada instância. Essas estimativas da magnitude das diferenças foram uti-

[4] A análise estatística dos resultados foi realizada do software estatístico R (R Development Core Team, 2011).

lizadas para avaliar o significado prático das diferenças entre os métodos. Em todos os casos, foi utilizado um nível de significância de 95%.

Como métrica de desempenho foi considerada a qualidade da solução final obtida pelos algoritmos, dada pelo valor da avaliação da função objetivo.

4.3 Análise dos resultados

4.3.1 Avaliação do algoritmo proposto em relação às outras adaptações do DE para otimização combinatória na solução do TSP

Para avaliar o algoritmo proposto (CoDE) em relação às adaptações já existentes do DE para otimização combinatória, foram consideradas três abordagens: lista de movimentos (LM), *relative position index* (RPI) e matriz de permutação (PM). As demais abordagens, matriz de adjacência, *smallest position value* e *forward/backward*, não foram utilizadas pois tendem a gerar muitas soluções inviáveis.

Os parâmetros dos algoritmos foram configurados da seguinte forma: (i) todos os algoritmos com o tamanho da população $n_p = 30$; (ii) os algoritmos CoDE, LM e PM foram configurados com o fator de ponderação $\psi = 1,0$ e; (iii) o algoritmo RPI foi configurado com $\psi = 0,5$ e $\rho = 0,9$.

O teste de significância realizado indicou a existência de diferenças significativas no desempenho médio dos algoritmos para as instâncias em geral ($p < 2 \times 10^{-16}$). A Figura 4.1 apresenta os estimadores de desempenho médio e intervalo de confiança ao longo de todas as instâncias.

Pode ser observado que o algoritmo proposto apresenta um desempenho muito superior em relação às demais abordagens baseadas no DE. Entre as abordagens por lista de movimentos e *relative position index* não foi detectada diferença significativa. No entanto, estas duas abordagens são superadas pela abordagem por matriz de permutação.

A partir das estimativas pontuais e intervalos de confiança para o desempenho dos algoritmos individualmente em cada instância, apresentados na Figura 4.2, é possível observar que a diferença de desempenho médio entre o CoDE e as demais abordagens baseadas em DE para esse problema se mantém consistente ao longo de todas as instâncias, e apresenta uma

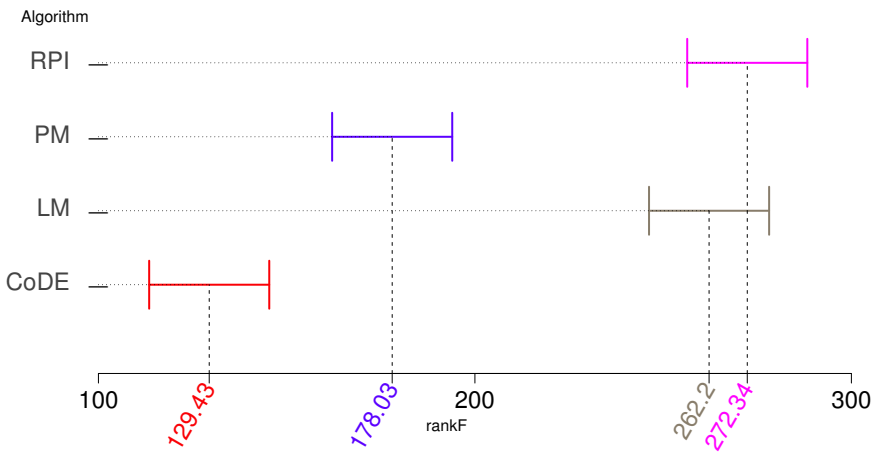


Figura 4.1 Intervalos de confiança para o desempenho geral dos algoritmos nas instâncias da base TSPLIB, após a remoção do efeitos devido às instâncias e interações algoritmo-instância. O algoritmo proposto (CoDE) obteve um desempenho bastante superior às demais abordagens testadas.

magnitude considerável em termos de qualidade das soluções retornadas.

Além disso, o CoDE foi capaz de obter valores médios bastante próximos dos ótimos de cada instância, diferentemente das três outras abordagens que retornaram soluções relativamente distantes do ótimo. Os valores na parte inferior da Figura 4.2 representam o número de execuções em que o algoritmo proposto alcançou o valor ótimo para as instâncias antes que o tempo limite fosse atingido. Pode-se observar que o CoDE foi capaz de encontrar o ótimo em todas as execuções para 13 das 21 instâncias utilizadas no experimento. Ele só não foi capaz de encontrar o ótimo para uma única instância, a *kroB150*. No entanto, mesmo para esta instância o desempenho médio obtido (26154,8) foi próximo ao ótimo (26130).

A Tabela 4.3 resume os resultados obtidos neste experimento. São apresentados a média e o desvio padrão para os valores das soluções obtidas, a taxa de convergência e o tempo médio até a convergência (considerando apenas as execuções em que o algoritmo foi capaz de convergir para a solução ótima).

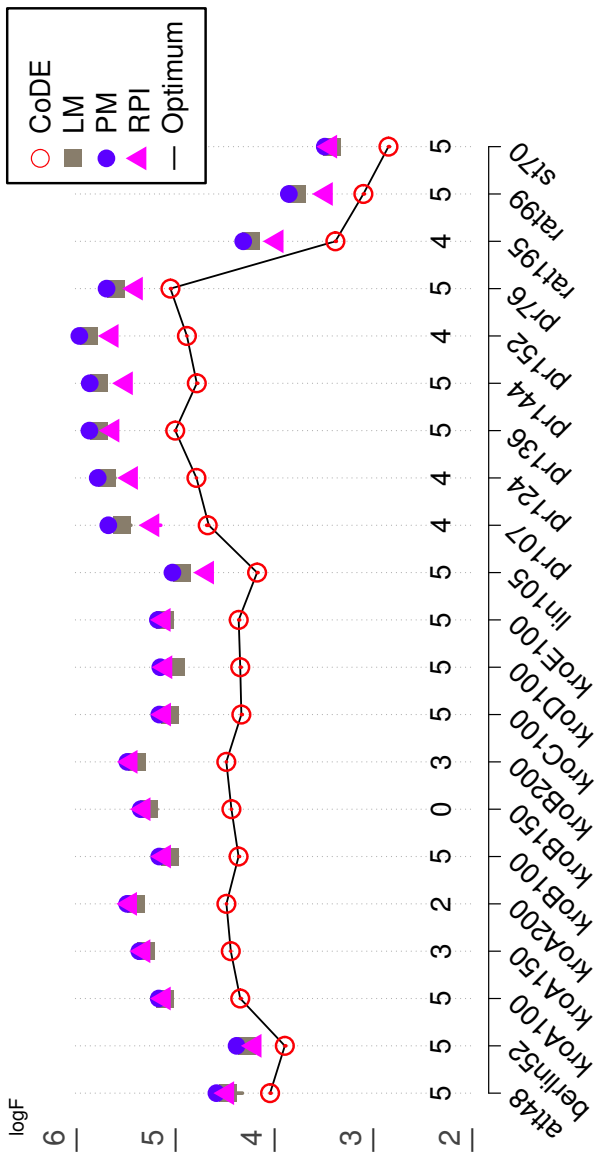


Figura 4.2 Desempenho médio dos algoritmos em cada instância da base TSPLIB, comparados ao valor ótimo (linha contínua). Os valores na parte inferior representam o número de execuções (até o máximo de 5) em que o algoritmo proposto (CoDE) alcançou o valor ótimo conhecido antes do limite de tempo. Nenhuma das outras abordagens foi capaz de encontrar a solução ótima em nenhuma instância dentro do limite de tempo de 1800 segundos. Os intervalos de confiança são representados por uma barra vertical sobre cada marcador. Na maior parte das instâncias os intervalos de confiança são menores que o tamanho do marcador utilizado.

Tabela 4.3 Resultados obtidos pelo algoritmo proposto para as instâncias da base TSPLIB

Instância	Ótimo	Média	Desv. Padrão	Convergência	Tempo médio (s)
att48	10628	10628,0	0,0	1,0	6,8
berlin52	7542	7542,0	0,0	1,0	3,9
kroA100	21282	21282,0	0,0	1,0	29,8
kroA150	26524	26531,0	15,1	0,6	259,0
kroA200	29368	29381,6	16,8	0,4	591,1
kroB100	22141	22141,0	0,0	1,0	36,7
kroB150	26130	26154,8	36,7	0,0	N/A
kroB200	29437	29447,8	18,6	0,6	1023,3
kroC100	20749	20749,0	0,0	1,0	178,9
kroD100	21294	21294,0	0,0	1,0	49,8
kroE100	22068	22068,0	0,0	1,0	416,0
lin105	14379	14379,0	0,0	1,0	33,3
pr107	44303	45475,4	2621,6	0,8	936,6
pr124	59030	59039,2	20,6	0,8	95,2
pr136	96772	96772,0	0,0	1,0	182,1
pr144	58537	58537,0	0,0	1,0	456,4
pr152	73682	73709,2	60,8	0,8	496,1
pr76	108159	108159,0	0,0	1,0	20,2
rat195	2323	2323,2	0,4	0,8	673,3
rat99	1211	1211,0	0,0	1,0	29,2
st70	675	675,0	0,0	1,0	12,3

4.3.2 Avaliação do algoritmo proposto em relação a abordagens não-populacionais na solução do CCCP

Após avaliar o desempenho do algoritmo proposto (CoDE) em relação às outras abordagens baseadas no DE para otimização combinatória, é avaliado o desempenho do CoDE em relação a abordagens não populacionais: um método exato utilizando o Gurobi (Gurobi Optimization Inc., 2014) que utiliza a estratégia *branch-and-bound*, aplicando métodos para otimização não-linear para resolver os subproblemas; uma heurística baseada no algoritmo *iterated local search* (ILS) (Lourenço *et al.*, 2010); e uma busca local *multi-start* utilizando soluções iniciais aleatórias (MultiStart).

O parâmetros para o CoDE foram mantidos iguais aos testes realizados para as instâncias do TSP. Os demais algoritmos não possuíam nenhum parâmetro adicional além do tempo

máximo de execução. O Gurobi foi configurado para utilizar uma única *thread* de processamento, mantendo seus demais parâmetros inalterados.

Os algoritmos CoDE, ILS e MultiStart foram executados 30 vezes para as 13 instâncias do CCCP. O Gurobi foi executado 30 vezes para cada uma das instâncias menores (família de instâncias *SJC*). Para a resolução das instâncias maiores (família de instâncias *Doni*), o Gurobi não foi capaz de encerrar sua execução dentro da restrição de tempo de 1800 segundos, extrapolando muito este limite.

A busca local utilizada nos algoritmos ILS e MultiStart faz uso do mesmo movimento de realocação utilizado pela busca local do CoDE, porém sem o limite de tempo. Como mecanismo de perturbação do ILS, é adotado o mesmo movimento de realocação, mas sem a verificação de qualidade da solução obtida, apenas a verificação da viabilidade da solução.

Os testes de significância realizados nos dois conjuntos de problemas (comparação das quatro abordagens no conjunto de instâncias *SJC*, e comparação do CoDE, ILS e MultiStart no conjunto de instâncias *Doni*) indicaram a presença de diferenças estatisticamente significativas ($p < 2 \times 10^{-16}$) no desempenho médio dos algoritmos.

O desempenho médio por instância e o desempenho médio geral, para o conjunto de instâncias *SJC*, são apresentados nas Figuras 4.3 e 4.4, respectivamente. É observado que o Gurobi obteve um desempenho muito inferior aos demais métodos em todas as instâncias dentro da restrição de tempo de 1800 segundos, enquanto o CoDE, ILS e MultiStart obtiveram um desempenho semelhante, com valores próximos aos melhores conhecidos na literatura.

Excepcionalmente para a instância *SJC3b*, o Gurobi apresentou algumas inconsistências em relação ao comportamento observado nas demais instâncias. Este comportamento diferente é devido à presença de dois *outliers* referentes a execuções do algoritmo que obtiveram resultados muito ruins. As razões que levaram a esta performance muito baixa do Gurobi não são conhecidas. Identificados os *outliers*, a média foi calculada sem a presença deles, o que é indicado na Figura 4.3 pelo marcador em cinza claro.

A estimativa do desempenho médio dos quatro algoritmos (após remoção dos efeitos devido às instância e interação algoritmo-instância) para as instâncias *SJC* em geral, apresentada na Figura 4.4, corrobora a conclusão de que as abordagens CoDE, ILS e MultiStart superam o Gurobi em qualidade da solução final obtida dentro da restrição de tempo utilizada. No entanto, entre o CoDE, ILS e MultiStart não foi detectada a presença de diferenças significativas.

A Figura 4.5 apresenta o desempenho médio dos algoritmos CoDE, ILS e MultiStart

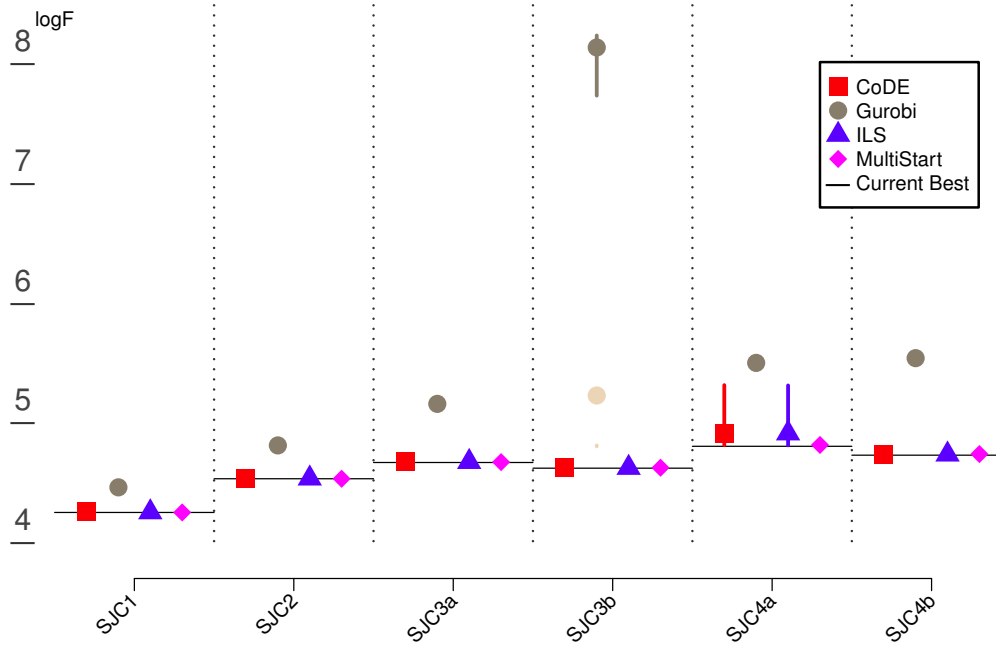


Figura 4.3 Desempenho médio dos algoritmos nas instâncias SJC. As barras na vertical representam o intervalo de confiança de 95% para o desempenho médio. O comportamento inconsistente do Gurobi para a instância *SJC3b* é devido a dois *outliers* em que a qualidade das soluções retornadas foi muito ruim. O marcador em cinza claro indica o desempenho médio do Gurobi após a remoção destes dois *outliers* dos dados.

para cada instância *Doni*. Neste conjunto de instâncias, o desempenho dos três algoritmos não apresentaram grandes variações, com exceção das instâncias menores *Doni1* e *Doni2*, em que o CoDE e ILS obtiveram um desempenho levemente melhor que o MultiStart. Nenhum dos três algoritmos foi capaz de obter valores próximos dos melhores resultados conhecidos dentro da restrição de tempo de 1800 segundos, com exceção do CoDE para a instância *Doni1*. Isso sugere a necessidade de um tempo maior de execução para que eles possam convergir.

A análise da estimativa do desempenho médio geral na Figura 4.6 reforça a conclusão de que a magnitude das diferenças para estas instâncias foram pequenas. Em particular, as diferenças entre o ILS e o CoDE, apesar de estatisticamente significativas, são muito pequenas na prática.

A Tabela 4.4 apresenta um resumo dos resultados obtidos, apresentando o melhor resultado encontrado, a média e o desvio padrão de cada algoritmo em cada instância utilizada nos experimentos. Os valores muito altos apresentados para o Gurobi na instância *SJC3b* são devidos aos dois *outliers* em que a qualidade das soluções retornadas foi muito ruim.

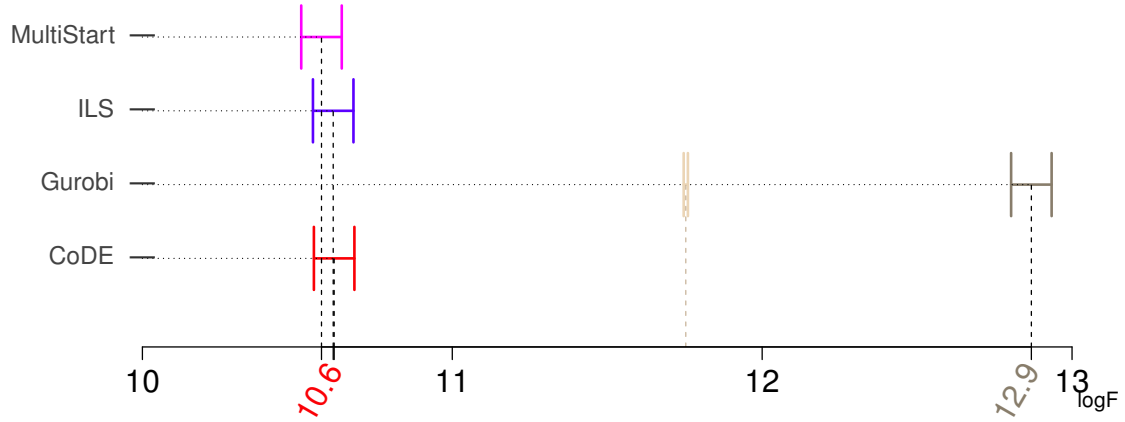


Figura 4.4 Intervalo de confiança para o desempenho geral dos algoritmos nas instâncias SJC, após a remoção do efeitos devido às instâncias e interações algoritmo-instância. Não foram detectados diferenças significativas entre os algoritmos CoDE, ILS e MultiStart. No entanto, esses três superaram o Gurobi com uma grande margem, mesmo após a remoção dos *outliers* (apresentado em cinza claro).

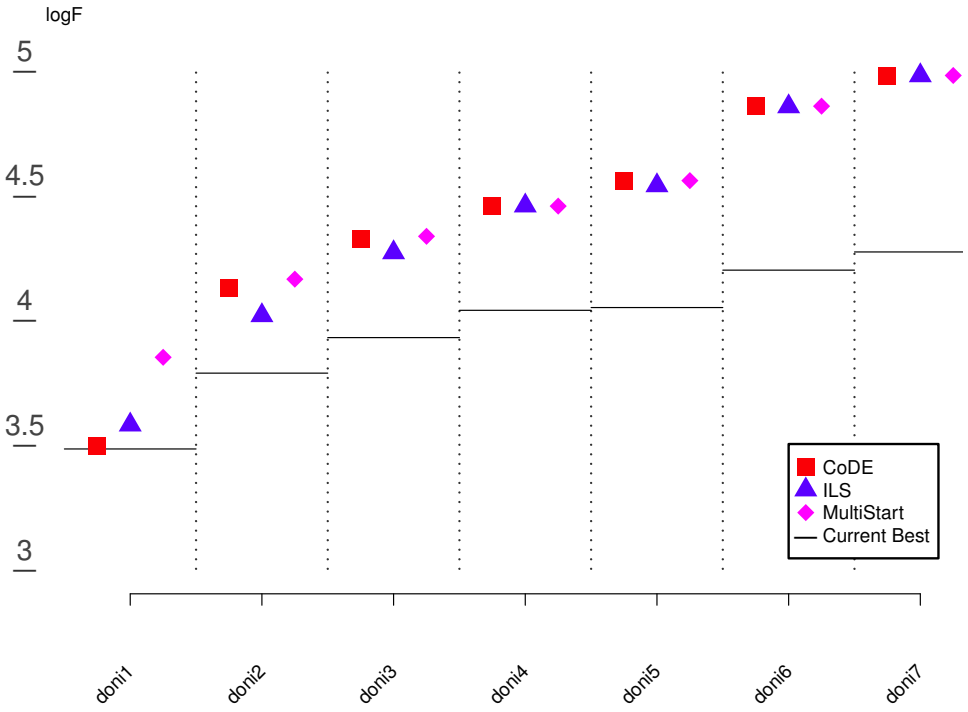


Figura 4.5 Desempenho médio dos algoritmos nas instâncias Doni. Foram calculados os intervalos de confiança de 95% para o desempenho médio, porém eles foram menores que o tamanho do marcador utilizado, não aparecendo no gráfico. Não é observado nenhuma diferença grande no desempenho pode ser observada, com exceção das duas instâncias menores (Doni1 e Doni2), onde o CoDE e ILS apresentaram, respectivamente, melhores resultados comparados aos outros métodos.

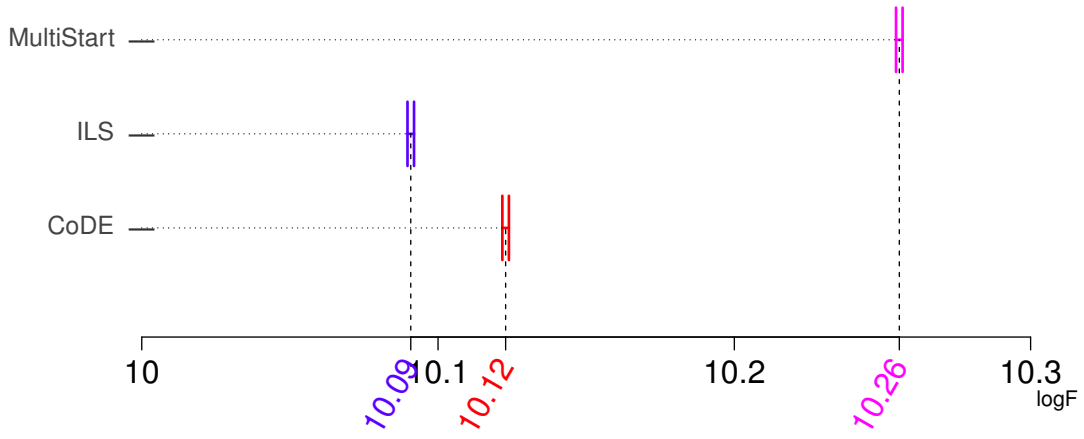


Figura 4.6 Intervalos de confiança para o desempenho global dos algoritmos nas instâncias Doni, após a remoção dos efeitos devidos às instâncias e interações algoritmo-instância. Apesar de estatisticamente significativa, a magnitude das diferenças observadas são pequenas na prática, particularmente entre o ILS e o CoDE.

Tabela 4.4 Resumo dos resultados obtidos pelos algoritmos para os conjuntos de instância SJC e Doni

		CoDE			Gurobi		
Instância	Melhor conhecido	Melhor	Média	Desv. Padrão	Melhor	Média	Desv. Padrão
SJC1	17359,75	17403,21	17607,72	152,24	27747,64	28198,12	177,04
SJC2	33181,65	33181,65	33385,29	128,19	62986,62	62986,62	0,00
SJC3a	45358,23	45646,94	45794,37	94,96	140011,85	140011,85	0,00
SJC3b	40661,94	40896,10	41120,03	107,55	164491,71	166101807,00	30688900,00
SJC4a	61931,60	62528,95	63331,35	116,56	307829,63	307829,63	0,00
SJC4b	52214,55	52692,81	52946,19	117,65	315022,58	315674,32	30678,70
Doni1	3021,41	3036,72	3118,19	29,99	N/A	N/A	N/A
Doni2	6080,70	12675,14	13404,07	252,53	N/A	N/A	N/A
Doni3	8438,96	20684,67	20989,17	120,29	N/A	N/A	N/A
Doni4	10854,48	28416,04	28444,01	9,46	N/A	N/A	N/A
Doni5	11134,94	35591,44	35655,29	20,74	N/A	N/A	N/A
Doni6	15722,67	71144,26	71195,01	19,21	N/A	N/A	N/A
Doni7	18596,74	94552,91	94613,54	18,81	N/A	N/A	N/A

		ILS			MultiStart		
Instância	Melhor conhecido	Melhor	Média	Desv. Padrão	Melhor	Média	Desv. Padrão
SJC1	17359,75	17359,75	17363,47	105,77	17359,75	17363,47	4,19
SJC2	33181,65	33181,65	33329,01	213,65	33181,65	33186,17	9,08
SJC3a	45358,23	**45356,37***	45703,58	505,38	45540,42	45703,43	87,03
SJC3b	40661,94	40670,90	40913,31	208,08	40847,47	40997,07	81,12
SJC4a	61931,60	62185,53	62995.59	572.33	62856,20	63555,02	290,22
SJC4b	52214,55	52309,56	52846,03	384,68	52811,21	53385,79	244,54
Doni1	3021,41	3635,22	3778,73	56,83	7011,51	7038,72	10,71
Doni2	6080,70	9880,75	10305,57	193,99	14451,90	14470,44	8,40
Doni3	8438,96	17669,29	18438,05	319,31	21461,33	21481,85	10,75
Doni4	10854,48	28456,49	28478,64	10,22	28365,18	28385,22	8,58
Doni5	11134,94	33972,88	34330,00	129,61	35875,37	35902,42	15,11
Doni6	15722,67	70990,08	71077,05	45,17	71284,78	71350,26	25,66
Doni7	18596,74	94542,81	94634,79	37,75	94660,21	94722,68	20,53

*** Novo valor para os melhores resultados conhecidos.**

4.4 Conclusões

O algoritmo proposto obteve um desempenho muito superior em relação às outras adaptações testadas do DE para otimização combinatória. As adaptações atualmente encontradas na literatura obtiveram valores muito distantes do valor ótimo, reforçando a afirmação de que os bons resultados reportados são devidos principalmente pelas técnicas de busca local utilizadas em conjunto com elas. Em contraste, o algoritmo proposto obteve valores muito próximos dos ótimos conhecidos, e efetivamente foi capaz de convergir para o ótimo na maior parte das execuções realizadas sobre as instâncias do TSP. Essa capacidade de convergência e exploração das imediações de uma dada solução pode ser potencialmente atribuída à estrutura do DE, que fornece ao algoritmo a capacidade de identificar as componentes do problema associadas a boas soluções no espaço de busca.

Ao resolver as instâncias menores do CCCP por um método exato baseado no *branch-and-bound* através do *solver* Gurobi, é notável a dificuldade desta abordagem em obter boas soluções, considerando a restrição de tempo de 1800 segundos. Essa dificuldade em obter boas soluções até mesmo para problemas menores é possivelmente devido à função objetivo não-linear. Dentro desta mesma restrição de tempo, o algoritmo proposto e as demais heurísticas baseadas em busca local apresentam resultados muito melhores.

Para instâncias menores do CCCP, o CoDE, ILS e MultiStart não apresentam diferenças significativas. Para instâncias maiores é possível observar uma separação entre estes três algoritmos. Embora os três algoritmos apresentem diferenças estatisticamente significativas, a magnitude dessas é pequena e, provavelmente, de baixo efeito prático. Isso é particularmente válido para a diferença de comportamento médio entre o ILS e o CoDE.

Esses resultados apontam que o CoDE, além de ser superior às atuais adaptações do DE para otimização combinatória, é capaz de obter resultados competitivos em relação a outras abordagens amplamente utilizadas na literatura para solução desta classe de problemas.

CAPÍTULO 5

Considerações Finais

Neste capitulo são feitas as considerações finais da dissertação, ressaltando os benefícios da codificação baseada em conjuntos que proporcionou ao algoritmo proposto um desempenho superior às demais adaptações do DE testadas. Também são apresentadas sugestões para continuidade do trabalho realizado.

5.1 Conclusões

O DE, em sua versão básica, é um algoritmo para solução de problemas de otimização com variáveis numéricas contínuas. No entanto, as adaptações encontradas na literatura desse algoritmo para problemas combinatórios não são capazes de preservar as características que o tornam uma poderosa ferramenta de otimização. A principal causa do baixo desempenho dessas adaptações é o uso de uma codificação ineficiente para uso com os operadores do DE, que tendem a não levar em consideração as características dos problemas combinatórios.

No presente trabalho foi introduzida uma codificação baseada em conjuntos para uso com a estrutura do DE, objetivando a solução de problemas de otimização combinatória em geral. O uso de conjuntos permite ao DE manipular informações do problema que são relevantes para que o espaço de busca possa ser explorado de maneira eficiente, embutindo uma certa inteligência nesta busca. A partir dos experimentos realizados, foi possível observar que o algoritmo proposto é capaz de preservar as características de convergência do DE original, através da utilização de operações sobre conjuntos capazes de definir regiões específicas do espaço nas quais uma busca local deve atuar. Através dessa estratégia, o algoritmo é capaz de identificar quais os componentes do problema combinatório são mais promissores a gerar soluções de melhor qualidade ao longo das iterações.

Além de preservar as características de convergência do DE, o algoritmo proposto é simples de configurar, requerendo o ajuste de poucos parâmetros. Além disso o método proposto é facilmente adaptável para outros problemas combinatórios, sendo necessário definir

basicamente dois aspectos do algoritmo: (i) quais os elementos manipulados pelo algoritmo; e (ii) qual a estratégia para a solução dos subproblemas.

Um ponto muito importante é a definição de quais elementos do problema serão manipulados pelo algoritmo. Geralmente, esses elementos são derivados das variáveis binárias do modelo matemático do problema, pois na maior parte dos problemas combinatórios são essas variáveis que detém as informações mais relevantes para definição dos subproblemas. Com isso evita-se trabalhar com dados que não agregam nenhum conhecimento ao algoritmo, como por exemplo, os rótulos numéricos arbitrários utilizados nas outras abordagens baseadas no DE para solução do TSP.

Para o CCCP, um problema cuja solução se torna bastante difícil a partir de estratégias baseadas nos métodos exatos tradicionais devido à sua função objetivo não-linear, os resultados obtidos pelo algoritmo proposto são competitivos quando comparados a outras abordagens amplamente utilizadas, como o *iterated local search* (ILS). Além disso, a estrutura do DE permite ao algoritmo proposto ser facilmente codificado para utilizar recursos de processamento paralelo, o que pode lhe dar uma vantagem ao resolver problemas maiores.

A partir dos experimentos realizados em relação ao uso da codificação de conjuntos juntamente com a estrutura do DE, observa-se que esta estratégia para representação de soluções contribui para que o algoritmo seja capaz de identificar componentes promissores para obtenção de boas soluções. Assim, a codificação aqui apresentada pode ser explorada em outros algoritmos evolucionários, substituindo as codificações baseadas em rótulos arbitrários e convertendo operadores aritméticos em operações sobre conjuntos, de forma a obter uma estrutura de busca mais eficiente.

5.2 Trabalhos futuros

O uso de processamento paralelo não foi explorado neste trabalho. A estrutura do DE permite que o algoritmo seja facilmente implementado utilizando estes recursos, o que pode lhe dar alguma vantagem em relação aos outros algoritmos na solução de instâncias maiores, tanto em termos de ganhos de desempenho como ganhos algorítmicos (p.ex., modelo de ilhas). Com isso, trabalhos futuros podem ser realizados a fim de avaliar o desempenho e o comportamento do algoritmo proposto em condições de execução paralela.

A partir do momento em que o algoritmo proposto constrói os subproblemas com me-

nos variáveis, o número de soluções candidatas é reduzido em relação ao problema original, porém a complexidade não é alterada. Assim, podem ser avaliadas outras estratégias de hibridização para que os subproblemas construídos sejam relaxações do modelo do problema original, reduzindo de fato a sua complexidade.

Além das questões apresentadas, um fator importante para algoritmos de otimização são os seus parâmetros de ajuste. Mesmo que o número de parâmetros seja pequeno, encontrar uma combinação de valores que proporcione um bom desempenho para todos os problemas não é uma tarefa tão simples. Os valores ideais podem variar de instância para instância, de acordo com suas características. Assim, estratégias de auto-adaptação dos parâmetros podem proporcionar um melhor desempenho do algoritmo, e constituem uma promissora linha de continuidade da pesquisa aqui apresentada.

Referências Bibliográficas

Addelman(1969) S. Addelman. The generalized randomized block design. *The American Statistician*, 23(4):35–36. Citado na pág. 45

Bean(1994) James C. Bean. Genetic algorithms and random keys for sequencing and optimization. *ORSA Journal on Computing*, 6(2):154–160. Citado na pág. 23

Belotti *et al.***(2013)** P. Belotti, C. Kirches, S. Leyffer, J. Linderoth, J. Luedtke, e A. Mahajan. Mixed-integer nonlinear optimization. *Acta Numerica*, 22:1–131. Citado na pág. 1, 33

Bertsimas e Tsitsiklis(1997) Dimitris Bertsimas e John Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1 edição. Citado na pág. 6, 10

Caserta e Voß(2010) Marco Caserta e Stefan Voß. Metaheuristics: Intelligent problem solving. Em Vittorio Maniezzo, Thomas Stützle, e Stefan Voß, editors, *Matheuristics: hybridizing metaheuristics and mathematical programming*, volume 10 of *Annals of Information Systems*, chapter 1, páginas 1–38. Springer US. Citado na pág. 1

Chaves e Lorena(2011) Antonio Augusto Chaves e Luiz Antonio Nogueira Lorena. Hybrid evolutionary algorithm for capacitated centered clustering problem. *Expert Systems with Applications*, 38:5013–5018. Citado na pág. 43

Crawley(2007) M. Crawley. *The R Book*. John Wiley & Sons, 1st edição. Citado na pág. 45

Davison e Hinkley(1997) A.C. Davison e D.V. Hinkley. *Bootstrap methods and their application*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press. Citado na pág. 45

Du e Pardalos(1999) Ding-Zhu Du e Panos M. Pardalos, editors. *Handbook of Combinatorial Optimization: Supplement*, volume A. Kluwer Academic Publishers. Citado na pág. 1

Ehrgott e Gandibleux(2000) Matthias Ehrgott e Xavier Gandibleux. A survey and annotated bibliography of multiobjective combinatorial optimization. *OR Spektrum*, 22:425–460. Citado na pág. 11

Garfinkel(1985) R. S. Garfinkel. *Traveling Salesman Problem: Motivation and Modeling*, chapter 2, páginas 17–36. Discrete Mathematics and Optimization. John Wiley & Sons. Citado na pág. 7

Gendreau e Potvin(2010) Michel Gendreau e Jean-Yves Potvin. Tabu search. Em Michel Gendreau e Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, chapter 2, páginas 41–59. Springer. Citado na pág. 12

Glover(1986) Fred Glover. Future path for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549. Citado na pág. 12

Glover e Kochenberger(2003) Fred W. Glover e Gary A. Kochenberger. *Handbook of Metaheuristics*. Springer, 1 edição. Citado na pág. 11

Goldbarg e Luna(2005) Marco Cesar Goldbarg e Henrique Pacca L. Luna. *Otimização combinatória e programação linear: modelos e algoritmos*. Elsevier, 2 edição. Citado na pág. 9, 10

Gurobi Optimization Inc.(2014) Gurobi Optimization Inc. Gurobi optimizer reference manual. http://www.gurobi.com/, 2014. Citado na pág. 41, 49

Hansen *et al.***(2010)** Pierre Hansen, Nenad Mladenović, Jack Brimberg, e José A. Moreno Pérez. Variable neighborhood search. Em Michel Gendreau e Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, chapter 3, páginas 61–86. Springer. Citado na pág. 12

Kirkpatrick *et al.***(1983)** S. Kirkpatrick, C. D. Gelatt, e M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680. Citado na pág. 12

Lichtblau(2009) Daniel Lichtblau. Relative position indexing approach. Em Godfrey C. Onwubolu e Donald Davendra, editors, *Differential Evolution: A Handbook for Global Permutation-Based Combinatorial Optimization*, volume 175 of *Studies in Computational Intelligence*, páginas 81–120. Springer. Citado na pág. 22

Lorena(2013) Luiz Antonio Nogueira Lorena. Instances for the capacitated centered clustering problem. http://www.lac.inpe.br/ lorena/antonio/CCCP.zip, 2013. Acessado em Dezembro 17, 2013. Citado na pág. 43

Lourenço *et al.***(2003)** Helena R. Lourenço, Olivier C. Martin, e Thomas Stützle. *Iterated local search*, chapter 11, páginas 321–353. International Series in Operational Research & Management Science. Kluwer Academic Publishers. Citado na pág. 12

Lourenço *et al.***(2010)** Helena R. Lourenço, Olivier C. Martin, e Thomas Stüzle. Iterated local search: framework and applications. Em Michel Gendreau e Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, chapter 12, páginas 363–397. Springer. Citado na pág. 12, 49

Lübbecke e Desrosiers(2005) Marco Lübbecke e Jacques Desrosiers. Selected topics in column generation. *Operations Research*, 53(6):1007–1023. Citado na pág. 9

Lust(2010) Thibaut Lust. *New metaheuristics for solving MOCO problems: application to the knapsack problem, the traveling salesman problem and IMRT optimization*. Tese de Doutorado, Université de Mons. Citado na pág. 12

Maniezzo *et al.***(2010)** Vittorio Maniezzo, Thomas Stützle, e Stefan Voß, editors. *Matheuristics - Hybridizing Metaheuristics and Mathematical Programming*, volume 10 of *Annals of Information Systems*. Springer. Citado na pág. 10, 11

Martello e Toth(1990) Silvano Martello e Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc. Citado na pág. 6

Michalewicz e Fogel(2004) Zbigniew Michalewicz e David B. Fogel. *How to Solve It: Modern Heuristics*. Springer Berlin Heidelberg, 2 edição. Citado na pág. 11, 12

Mladenović e Hansen(1997) N. Mladenović e P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100. Citado na pág. 12

Montgomery(2008) D. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons. Citado na pág. 45

Moraglio e Poli(2004) Alberto Moraglio e Riccardo Poli. Topological interpretation of crossover. Em *Proccedings of the Genetic and Evolutionary Computation Conference*, páginas 1377–1388. Citado na pág. 32

Moraglio e Poli(2011) Alberto Moraglio e Riccardo Poli. Geometric crossover for the permutation representation. *Intelligenza Artificiale*, 5:49–63. Citado na pág. 32

Negreiros e Palhano(2006) Marcos Negreiros e Augusto Palhano. The capacitated centered clustering problem. *Computers & Operations Research*, 33:1639–1663. Citado na pág. 41, 42

Neumann e Witt(2010) Frank Neumann e Carsten Witt. *Bioinspired Computation in Combinatorial Optimization: Algorithms and Their Computational Complexity*. Natural computing series. Springer. Citado na pág. 13

Nikolaev e Jacobson(2010) Alexander G. Nikolaev e Sheldon H. Jacobson. Simulated annealing. Em Michel Gendreau e Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, chapter 1, páginas 1–39. Springer. Citado na pág. 12

Onwubolu(2001) Godfrey Onwubolu. Optimization using differential evolution algorithm. Relatório técnico, IAS. Citado na pág. 23

Onwubolu e Davendra(2006) Godfrey Onwubolu e Donald Davendra. Scheduling flow shops using differential evolution algorithm. *European Journal of Operational Research*, 171(2): 674–692. Citado na pág. 13

Onwubolu e Davendra(2009a) Godfrey Onwubolu e Donald Davendra. Motivation for differential evolution for permutative-based combinatorial problems. Em Godfrey C. Onwubolu e Donald Davendra, editors, *Differential Evolution: A Handbook for Global Permutation-Based Combinatorial Optimization*, volume 175 of *Studies in Computational Intelligence*, chapter 1, páginas 1–11. Springer. Citado na pág. 13, 17

Onwubolu e Davendra(2009b) Godfrey Onwubolu e Donald Davendra. Differential evolution for permutation-based combinatorial problem. Em Godfrey C. Onwubolu e Donald Davendra, editors, *Differential Evolution: a handbook for global permutation-based combinatorial optimization*, volume 175 of *Studies in Computational Intelligence*, chapter 2, páginas 13–34. Springer. Citado na pág. 7

Onwubolu e Davendra(2009c) Godfrey C. Onwubolu e Donald Davendra, editors. *Differential Evolution: a handbook for global permutation-based combinatorial optimization*, volume 175 of *Studies in Computational Intelligence*. Springer. Citado na pág. 13

Pan *et al.***(2008)** Quan-Ke Pan, Mehmet Fatih Tasgetiren, e Yun-Chia Liang. A discrete differential evolution algorithm for the permutation flowshop scheduling problem. *Computers & Industrial Engineering*, 55(4):795–816. Citado na pág. 13

Prado *et al.***(2010)** Ricardo S. Prado, Rodrigo C. P. Silva, Frederico G. Guimarães, e Oriane M. Neto. Using differential evolution for combinatorial optimization: a general approach. Em

2010 IEEE International Conference on Systems Man and Cybernetics, páginas 11–18. Citado na pág. 2, 13, 17, 25, 26

Price *et al.*(**2005**) Kenneth Price, Rainer M. Storn, e Jouni Lampinen. *Differential Evolution: A Practical Approach to Global Optimization*. Natural Computing Series. Springer. Citado na pág. 7, 17, 18, 19, 20, 22, 25

R Development Core Team(2011) R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011. URL http://www.R-project.org/. Citado na pág. 45

Raidl e Julstrom(2003) G. R. Raidl e B. A. Julstrom. Edge sets: An effective evolutionary coding of spanning trees. *IEEE Transactions on Evoutionary Computation*, 7(3):225–239. Citado na pág. 2

Rao(2009) Singiresu S. Rao. *Engineering optimization: theory and practice*. John Wiley & Sons, 4 edição. Citado na pág. 6

Reinelt(1995) Gerhard Reinelt. TSPLIB 95. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/DOC.PS, 1995. Acessado em Dezembro 17, 2013. Citado na pág. 19, 23, 26, 35, 40

Resende e Ribeiro(2010) Mauricio G.C. Resende e Celso C. Ribeiro. Greedy randomized adaptive search procedures: Advances, hybridizations, and applications. Em Michel Gendreau e Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, chapter 10, páginas 283–319. Springer. Citado na pág. 12

Rothlauf e Tzschoppe(2005) Franz Rothlauf e Carsten Tzschoppe. Making the edge-set encoding fly by controlling the bias of its crossover operator. Em *Proc. 5th European Conference on Evolutionary Computation in Combinatorial Optimization*, EvoCOP'05, páginas 202–212, Berlin, Heidelberg. Springer-Verlag. Citado na pág. 2

Sauer *et al.*(**2011**) João Guilherme Sauer, Leandro Santos Coelho, Viviana Cocco Mariani, Luiza Macedo Mourelle, e Nadia Nedjah. A discrete differential evolution approach with local search for traveling salesman problems. Em Nadia Nedjah, Leandro Santos Coelho, Viviana Cocco Mariani, e Luiza Macedo Mourelle, editors, *Innovative Computing Methods and Their Applications to Engineering Problems*, volume 357 of *Studies in Computational Intelligence*, páginas 1–12. Springer Berlin Heidelberg. Citado na pág. 13

Stefanello e Müller(2009) Fernando Stefanello e Felipe Martins Müller. Um estudo sobre problemas de agrupamento capacitado. Em *Anáis do XLI Simpósio Brasileiro de Pesquisa Operacional*. Citado na pág. 42

Storn(2008) Rainer Storn. Differential evolution research - trends and open questions. Em Uday K. Chakraborty, editor, *Advances in Differential Evolution*, volume 143 of *Studies in Computational Intelligence*, páginas 1–31. Springer Berlin Heidelberg. Citado na pág. 2, 17, 28

Storn e Price(1995) Rainer Storn e Kenneth Price. Differential evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces. Relatório técnico, International Computer Science Institute. Citado na pág. 2, 13

Storn e Price(1997) Rainer Storn e Kenneth Price. Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11:341–359. Citado na pág. 2, 13, 15, 33

Tasgetiren *et al.***(2009)** Fatih Tasgetiren, Angela Chen, Gunes Gencyilmaz, e Said Gattoufi. Smallest position value approach. Em Godfrey C. Onwubolu e Donald Davendra, editors, *Differential Evolution: A Handbook for Global Permutation-Based Combinatorial Optimization*, volume 175 of *Studies in Computational Intelligence*, páginas 121–138. Springer. Citado na pág. 23

Weise(2009) Thomas Weise. *Global optimization algorithms: theory and applications*. Self-Published, 2 edição. URL `http://www.it-weise.de/`. Citado na pág. 10

Wolsey(1998) Laurence A. Wolsey. *Integer Programming*. Wiley Series in Discrete Mathematics and Optimization. Wiley. Citado na pág. 1, 5, 6, 7, 9, 10, 11, 33