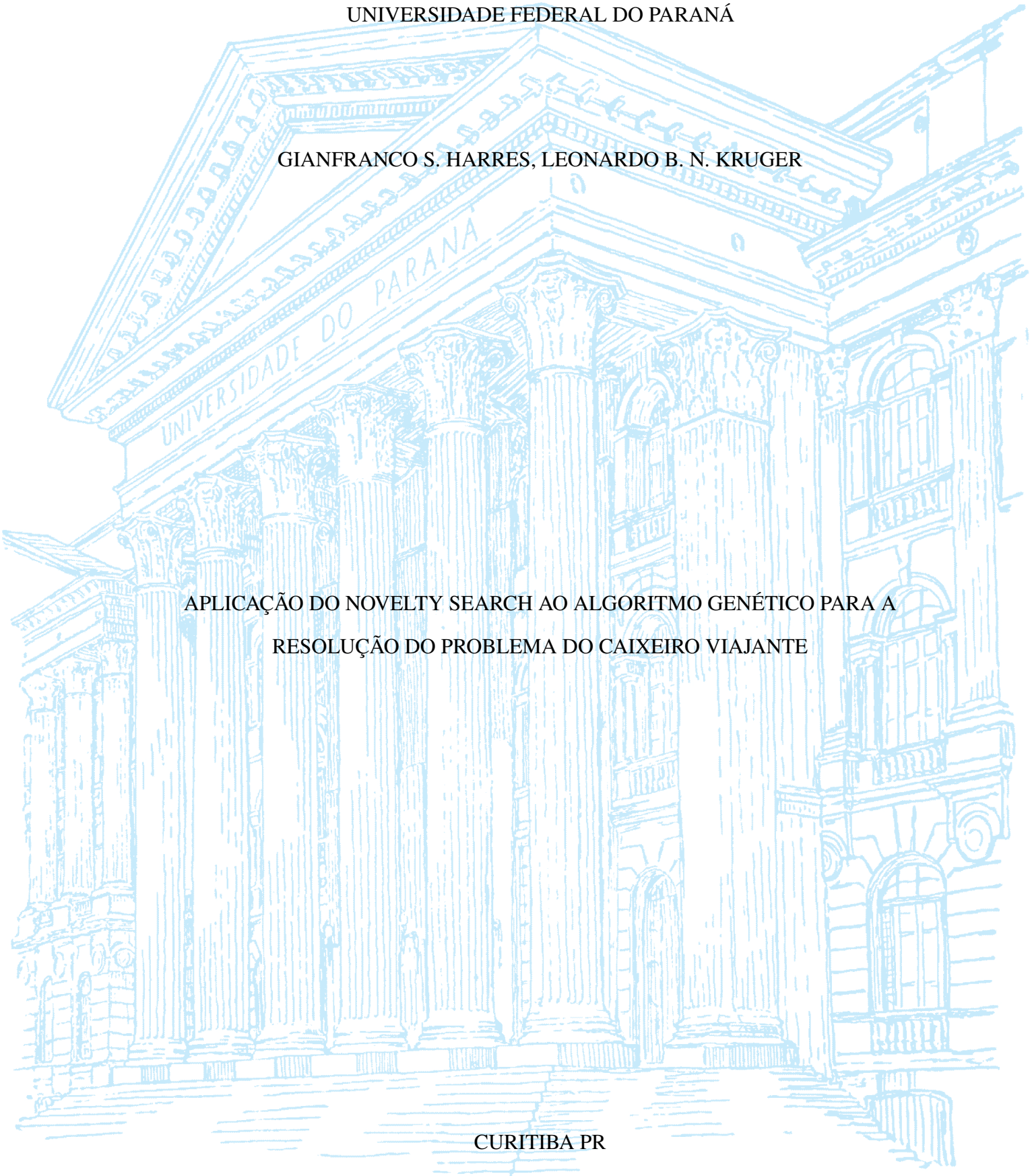
UNIVERSIDADE FEDERAL DO PARANÁ

GIANFRANCO S. HARRES, LEONARDO B. N. KRUGER



APLICAÇÃO DO NOVELTY SEARCH AO ALGORITMO GENÉTICO PARA A RESOLUÇÃO DO PROBLEMA DO CAIXEIRO VIAJANTE

CURITIBA PR

2023

GIANFRANCO S. HARRES, LEONARDO B. N. KRUGER

APLICAÇÃO DO NOVELTY SEARCH AO ALGORITMO GENÉTICO PARA A RESOLUÇÃO DO PROBLEMA DO CAIXEIRO VIAJANTE

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Eduardo Spinosa.

CURITIBA PR

2023

RESUMO

O Problema do Caixeiro Viajante (TSP) é um desafio clássico na área da Ciência da Computação, sendo um dos problemas mais estudados e ainda não é conhecido um algoritmo que resolva o problema em tempo polinomial. Diversas abordagens têm sido propostas, incluindo técnicas bioinspiradas, como o Algoritmo Genético (GA). Este trabalho apresenta o Algoritmo Genético, sua modelagem e aplicação para o TSP e uma abordagem utilizando o GA com a busca no espaço comportamental, chamado Novelty Search (NS). O NS se concentra na descoberta de soluções inovadoras e não convencionais, promovendo a diversidade e evitando a convergência prematura em ótimos locais. É realizado também um estudo comparativo entre os resultados do GA com os resultados da aplicação do NS ao GA, chamado GA_{NS}.

Palavras-chave: Problema do Caixeiro Viajante. Algoritmo Genético. Novelty Search. Otimização. Diversidade. Estagnação.

ABSTRACT

The Traveling Salesman Problem (TSP) is a classic challenge in the field of Computer Science, and it remains one of the most studied problems for which a polynomial-time algorithm has not yet been discovered. Various approaches have been proposed, including bioinspired techniques such as the Genetic Algorithm (GA). This paper presents the Genetic Algorithm, its modeling and application to the TSP, and an approach that combines the GA with the behavioral space search called Novelty Search (NS). NS focuses on discovering innovative and unconventional solutions, promoting diversity, and avoiding premature convergence to local optima. A comparative study is also conducted between the results of the GA and the results of applying NS to the GA, referred to as GA_{NS}.

Keywords: Traveling Salesman Problem. Genetic Algorithm. Novelty Search. Optimization. Diversity. Stagnation.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

DINF	Departamento de Informática
UFPR	Universidade Federal do Paraná
TSP	Traveling Salesman Problem
GA	Genetic algorithm
PSO	Particle Swarm Optimization
NS	Novelty Search
KNN	K-Nearest Neighbors
BA	Bat algorithm

SUMÁRIO

1 INTRODUÇÃO

Organismos vivos são muito reconhecidos pelas suas capacidades versáteis em resolução de problemas (6) através de habilidades provenientes de processos evolutivos e seleção natural. Essa observação tem inspirado muitos pesquisadores a buscar a replicação desses comportamentos por meio de algoritmos computacionais, com o objetivo de resolver problemas de alta complexidade.

Um exemplo de problema clássico é o chamado "Problema do Caixeiro Viajante"(ou TSP, do inglês "Traveling Salesman Problem")(7), que tem como objetivo encontrar a rota mais curta para visitar um conjunto de cidades sem repeti-las, retornando ao ponto de partida. O TSP é amplamente estudado na área da ciência da computação, devido à sua abstração de aplicações comerciais e sociais complexas, tais como logística e transporte. No entanto, o TSP é um problema NP-difícil, isso significa que, o TSP faz parte do conjunto de problemas de otimização combinatória para os quais não se conhecem soluções eficientes em tempo polinomial. Diante dessa dificuldade, muitos pesquisadores recorrem à Inteligência Artificial na busca por soluções para o TSP. Nesse contexto, este trabalho de graduação se concentra na busca de soluções para o TSP.

O TSP pode ser representado por meio da Teoria dos Grafos, em que cada cidade é um vértice e a distância entre cada par de cidades é uma aresta com peso. Sendo assim, uma solução para o TSP é uma permutação de cidades, onde o ponto de início é também o ponto final. Obter a melhor solução para o TSP equivale encontrar o ciclo de menor custo no grafo. Existem diversas técnicas que podem ser utilizadas para resolver esse problema descrito, como a busca em profundidade, busca em largura, árvores geradoras, técnicas como programação linear e algoritmos bio-inspirados.

Ao longo dos anos, muitas soluções foram propostas, e elas variam de acordo com a classe dos algoritmos utilizados. Algoritmos exatos, como o algoritmo Branch-and-bound, sempre encontram a melhor solução do problema ao realizar uma busca exaustiva por todo o espaço. No entanto, devido ao seu alto custo de execução, geralmente não é usado em problemas NP-difícil.

Algoritmos de Aproximação, como o algoritmo de Christofides e a Heurística do vizinho mais próximo, não encontram sempre a melhor solução, mas oferecem maior eficiência computacional. Eles são geralmente escolhidos para problemas em que encontrar a melhor solução é muito custoso.

Outra classe de soluções é a dos algoritmos baseados em meta-heurísticas, como o Algoritmo Genético (GA), que usa heurísticas e técnicas de otimização para encontrar uma solução de relativa boa qualidade com baixa dificuldade na modelagem para o problema. No entanto, há muita variação nos resultados de acordo com a modelagem das heurísticas, podendo levar o algoritmo a uma convergência prematura, levando a uma estagnação em ótimos locais, dessa forma, diminuindo a qualidade da solução.

Em 2011, (9) propuseram o conceito de Novelty Search (NS) como a solução para o problema fundamental do paradigma orientado a objetivos. Tal problema fundamental é descrito como a busca por otimizar um objetivo específico pode a levar a caminhos sem saída, chamados ótimos locais. Com o passar do tempo, o NS foi explorado em diversos artigos com um objetivo específico, solucionar problemas provenientes dos algoritmos baseados em população orientados a objetivo. A implementação do NS trouxe diversas vantagens, dentre elas estão a geração de

maior diversidade nas populações, exploração de espaços inexplorados, promoção de inovação nas soluções, robustez a cenários enganosos e alta capacidade de vencer ótimos locais.

Baseado nesses argumentos, esse trabalho propõe uma abordagem usando o Algoritmo Genético tradicional em conjunto com o Novelty Search e levantar os ganhos provenientes dessa combinação em relação ao Algoritmo Genético tradicional. Mais especificamente, nesse trabalho é realizada a implementação e aplicação do Algoritmo Genético tradicional ao TSP, a combinação do GA tradicional com o NS, o estudo e a implementação do NS de forma otimizada para o TSP e a comparação dos resultados com outras técnicas bioinspiradas para a resolução do problema.

No capítulo 2 é apresentado e definido o problema do TSP e a forma de solucioná-lo. É apresentado também o embasamento teórico para o Algoritmo Genético tradicional e para o Novelty Search. No capítulo 3 são apresentados trabalhos relacionados a esta proposta. No capítulo 4 é mostrada a modelagem do algoritmos e suas respectivas implementações para a resolução do TSP. Por fim, no capítulo 5 a metodologia para levantar os resultados é apresentada e os resultados obtidos experimentalmente.

2 FUNDAMENTAÇÃO

2.1 PROBLEMA DO CAIXEIRO VIAJANTE

O problema do Caixeiro Viajante (Traveling Salesman Problem - TSP) é um dos mais estudados e desafiadores problemas da ciência da computação (2). Embora pareça simples à primeira vista, a complexidade do TSP aumenta exponencialmente à medida que o número de cidades aumenta, tornando-o um caso clássico de estudo.

A resolução do TSP tem uma grande importância tanto teórica quanto prática. Do ponto de vista teórico, o TSP é um exemplo clássico de problema NP-difícil, que é um conjunto de problemas de otimização combinatória que não são conhecidas soluções em tempo polinomial. Esse fato o torna um objeto de estudo interessante na teoria da complexidade computacional e na matemática discreta.

Por outro lado, do ponto de vista prático, o TSP tem aplicações em diversas áreas, tais como logística, planejamento de rotas, sistemas de transporte, planejamento de circuitos integrados, entre outras. Em muitos casos, encontrar a solução ótima para o TSP é crucial para a tomada de decisões eficientes e para a redução de custos.

Aliado a esse fato, e além de possuir fácil entendimento, o TSP é tido como um benchmark padrão para novos algoritmos e uma boa prova de performance e efetividade de solução pela sua natureza de dificuldade e de aplicação a problemas do mundo real.

A resolução do TSP é um desafio computacional extremamente complexo, e ao longo dos anos, pesquisadores têm desenvolvido diversas soluções para enfrentar esse problema. As soluções vão desde algoritmos exatos, que garantem a solução ótima, mas têm alta complexidade computacional, até algoritmos de aproximação, que buscam soluções sub-ótimas com menor custo computacional.

Mais recentemente, técnicas de inteligência artificial e aprendizado de máquina têm sido utilizadas para lidar com o TSP, como algoritmos genéticos, redes neurais, e outras técnicas de otimização. Essas técnicas apresentam resultados promissores em termos de eficiência computacional e qualidade das soluções encontradas. E iremos abordar uma nova técnica seguindo essa ideia e utilizando GA e NS em conjunto.

2.1.1 Definição

O Problema do Caixeiro Viajante (TSP) pode ser conceituado como um conjunto de cidades representadas em um grafo, no qual as conexões entre cidades são representadas por arestas com pesos que indicam os custos de deslocamento entre elas. O objetivo é encontrar o caminho de custo mínimo que visite cada cidade exatamente uma vez, formando o que é conhecido como um Caminho Hamiltoniano. Em termos simples, um Caminho Hamiltoniano em um grafo é uma sequência de vértices que percorre todos os vértices do grafo uma única vez, seguindo as arestas do grafo, sem repetições. Não é necessário que o caminho comece e termine no mesmo vértice. No entanto, para atender aos requisitos do TSP, o ponto de partida deve ser o mesmo que o ponto de chegada, criando assim um Ciclo Hamiltoniano. Portanto, uma solução para o TSP pode ser representada por uma permutação das cidades, onde o ponto de início e o ponto final são os mesmos. Encontrar a melhor solução para o TSP equivale a encontrar um ciclo de custo mínimo no grafo que visita cada cidade exatamente uma vez.

A definição matemática do TSP definida em (5) ou (13) consiste em obter uma lista de n cidades $C = (c_1, c_2, ..., c_i, ..., c_j, ..., c_n)$ e a matriz de distância $D = (d_{i,j})_{n \times n}$, onde $d_{i,j}$

representa a distância Euclidiana da cidade c_i para a cidade c_j. Nesse caso, é adotado a navegação em plano, com a distância Euclidiana definida em 2.1.

$$E(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{2.1}$$

Com as distâncias (x_i, y_i) representando as coordenadas de c_i e (x_j, y_j) as coordenadas de c_j. Se para todo $c_i, c_j \in C$, com $i! = j$, as distâncias $d_{i,j} = d_{j,i}$ o problema é dito simétrico.

O objetivo é encontrar o menor tour t em $T = (t_1, t_2, ..., t_n)$ onde T é todo tour possível e $t = (t_{k1}, t_{k2}, ..., t_{kn})$ uma permutação de C. Sendo assim, o menor tour t é aquele que minimize a função custo 2.2

$$F(t) = \sum_{i=1}^{n-1} d_{t(k),t(k+1)} + d_{t(n),t(1)} \tag{2.2}$$

2.2 ALGORITMO GENÉTICO

O Algoritmo Genético, desde sua primeira aplicação em 1975 (20), se tornou uma das técnicas mais aplicadas para resolução de problemas otimização combinatorial (Osaba et al.), é uma meta-heurística baseada no processo de seleção natural da evolução biológica. A ideia é que, assim como as espécies evoluem para se adaptar ao meio ambiente ao longo do tempo, uma população de soluções candidatas para um problema pode evoluir para encontrar uma solução ótima (1).

Nesse contexto, o processo de evolução no Algoritmo Genético pode ser interpretada como uma população inicial de soluções candidatas obtidas de forma aleatória onde cada solução representa um indivíduo. Cada indivíduo possui suas próprias características, que são seus genes, consistindo tais genes de combinações e recombinações de seus antecessores. Tal população, com o passar do tempo, passa por diversos processos de seleção onde as soluções mais aptas sobrevivem, de forma semelhante à seleção natural, e convergem para uma solução melhor do problema.

Com o passar das gerações, as soluções sobreviventes possuem a chance de se reproduzirem, levando suas características para as gerações posteriores. Esse processo é chamado de evolução e possui dois operadores no Algoritmo Genético, o crossover e a mutação. Tais operadores são responsáveis por adicionar diversidade nas soluções. O crossover, simula o processo de reprodução sexual combinando os genes de dois indivíduos, chamados parentes, gerando um ou mais indivíduos, chamados filhos, que possuem as características de ambos os geradores. Nessa mesma linha, a mutação simula a mudança aleatória proveniente das mutações espontâneas em genes na natureza. Essa operação permite a introdução de características diferentes na população, promovendo uma maior diversidade não dependente de indivíduos geradores. O processo de evolução ocorre diversas vezes até que um critério de parada seja atingido, como um número máximo de gerações ou uma convergência ótima seja alcançada.

Os algoritmos genéticos podem ser aplicados a uma ampla variedade de problemas de otimização, incluindo problemas de roteamento, de agendamento, de design e muitos outros. Eles são particularmente úteis em casos em que o espaço de busca é muito grande para ser explorado de forma exaustiva, ou em que a função objetivo não é diferenciável ou possui muitos máximos locais.

Apesar de ser uma técnica poderosa que permite uma fácil aplicação a diversos problemas, o Algoritmo Genético possui muitos parâmetros e ajustes que precisam ser otimizados para cada problema específico para encontrar a melhor eficiência, o que pode ser uma tarefa complexa.

Além disso, há o risco de a população ficar presa em um ótimo local. O problema do ótimo local ocorre quando um algoritmo encontra uma solução que é a melhor dentro de uma determinada vizinhança, mas essa solução não é necessariamente a melhor solução global e a exploração não obteve sucesso em encontrar a melhor solução possível para todo o espaço de busca. No entanto, quando aplicados corretamente, os algoritmos genéticos podem fornecer soluções eficientes e eficazes para uma ampla variedade de problemas de otimização.

2.2.1 Definição

Em (22) define o algoritmo genético como um conjunto de tarefas a ser concluídas. Essas tarefas são o núcleo do GA e consistem de codificação genética, função fitness, operador genético (crossover e mutação), parâmetros do algoritmo (inicialização de população e parâmetros) e condição de finalização. Com isso, os passos e heurísticas para modelar um algoritmo genético podem ser definidos da seguinte maneira:

- `Codificação`: Refere-se a como os cromossomos representam o problema. Pode ser definido de diversas maneiras e são dependentes da natureza do problema. É possível utilizar números, permutações ou grafos como codificação dos cromossomos.

  A permutação, comumente utilizado em problemas de roteamento, possui cada gene do cromossomo representando um elemento e a ordem em que cada gene aparece no cromossomo define a qualidade da solução. Como todos os cromossomos possuem os mesmos genes, alterar a ordem de um gene gera um novo cromossomo. Sendo assim, para um problema com k genes, onde todos os k devem aparecer sem repetição, existem $k!$ cromossomos possíveis. Uma má codificação do problema pode tornar o algoritmo com um péssimo desempenho ou tornar impossível de solucioná-lo.

  Com isso, a codificação é uma etapa importante para definir a abordagem do problema por possuir impacto direto no espaço de busca e em como as demais etapas interagirão com o problema.

- `Função fitness`: Crucial para o andamento do algoritmo, define a avaliação da qualidade das soluções, necessária para as demais funções do algoritmo. A função fitness é definida por um conjunto de regras, dependente do problema, em que aplicada gera o fitness de um indivíduo. O fitness, qualifica numericamente uma solução baseada na aptidão do cromossomo, permitindo ao algoritmo distinguir soluções potencialmente boas de soluções ruins, guiando o algoritmo pelo espaço de busca em direção a solução ideal do problema. Esse direcionamento acontece através da seleção. Porém, a seleção possui suas próprias particularidades e modos de realizar o direcionamento, mas é fundamentalmente dependente do fitness de cada indivíduo. Sendo assim, uma má aplicação do fitness pode levar o algoritmo a não convergência ao aplicar erroneamente um fitness alto a um péssimo candidato, ou vice-versa.

- `Seleção`: Responsável diretamente pela convergência, diversidade e exploração do espaço. Consiste da simulação do processo de seleção natural, onde é responsável em selecionar indivíduos através de uma métrica pré-estabelecida. Isso garante que os melhores indivíduos terão maior probabilidade de sobreviverem para a próxima geração e de passar seus traços para seus filhos.

  Para o contexto de seleção, diversas métricas e métodos podem ser utilizadas e variam com o problema, como o número de indivíduos que podem reproduzir, modos de seleção

de indivíduos, ranqueamento e elitismo. Pode ser utilizada mais de uma métrica para definir o candidatos a reprodução e deve ser utilizada de forma que seja possível as gerações caminharem em direção da melhor solução.

- `Crossover`: Em paralelo ao processo de recombinação genética, crossover é o processo que combina dois ou mais indivíduos. O resultado do crossover gera um ou mais novos indivíduos, chamados filhos, onde cada filho contém os traços de seus parentes. Esse processo é muito importante nos algoritmos genéticos, já que é o principal método para encontrar indivíduos superiores. Partindo do ponto de que o objetivo do crossover é obter melhores soluções a partir das soluções boas já obtidas, nesse trabalho é utilizado o método de crossover mapeado, onde o crossover de um individuo é feito com seu vizinho.

- `Mutação`: Processo baseado na mutação, onde um gene de um indivíduo pode ser aleatoriamente modificado, produzindo assim uma nova variante. Muito importante na biologia para manter a diversidade da população, podendo aumentar ou diminuir sua adaptabilidade ao ambiente. Similarmente, no contexto da aplicação ao problema, a mutação mantém a diversidade das soluções, evitando ótimos locais. Essa mutação acontece a partir de uma probabilidade, que quando satisfeita, ocorre a mudança de um gene do individuo. Nesse caso, a mutação assume a responsabilidade de aumentar a busca local do algoritmo e prevenir travamentos em ótimos locais. Sendo assim, uma probabilidade baixa de mutação pode levar a uma baixa diversidade de soluções e quando muito alta pode levar a destruição completa de um gene, deteriorando a convergência.

- `Condição de Finalização`: Mesmo o GA sendo um algoritmo bioinspirado, onde, na natureza, não existiria uma condição de finalização, é necessário que exista uma na modelagem para se obter um algoritmo factível para o mundo real. Tal condição é chamada de condição de finalização. Essa condição é a que define quando o algoritmo deve parar e devolver o melhor resultado. Uma má modelagem da condição de finalização pode levar a introdução de convergência prematura e a não localização do ótimo global. Com isso, a condição de finalização precisa balancear bem o custo computacional com a exploração do espaço. Neste contexto, a única condição de finalização escolhida é o número de gerações, deixando o custo computacional medido pela velocidade de convergência para o melhor resultado.

Todas as etapas são importantes e devem ser modeladas com muita atenção as restrições do problema. Cada etapa impacta diretamente a capacidade de exploração do espaço e o andamento das demais etapas, consequentemente, impacta na qualidade final e tempo de convergência do algoritmo.

A figura 2.1 mostra graficamente o funcionamento do GA e como suas etapas se conectam para resolver um problema. A etapa de início, é responsável pela inicialização e codificação da primeira população. Após isso, é iniciado o fluxo do algoritmo, onde a condição de finalização é verificada. Dentro do ciclo do algoritmo, são realizadas o cálculo do fitness dos indivíduos, a seleção baseada no fitness, os operadores genéticos crossover e mutação aplicados nos indivíduos selecionados, formando assim uma nova população, simbolizando o início de uma nova geração.

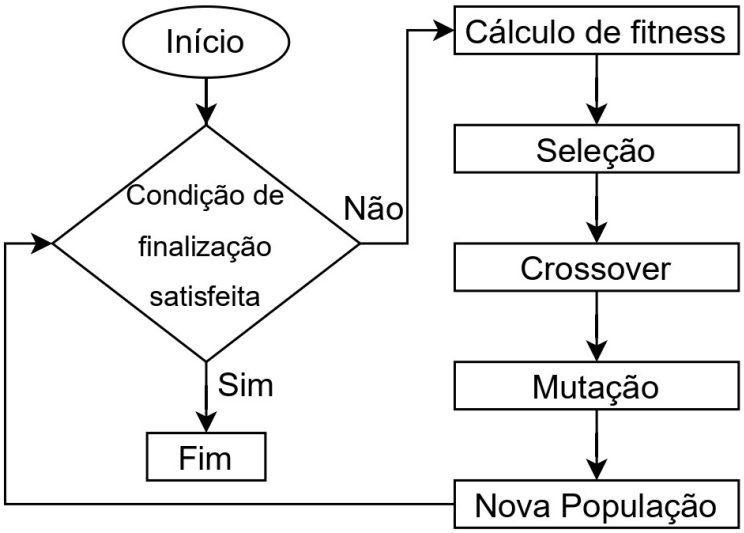


Figura 2.1: Fluxograma com etapas realizadas pelo Algoritmo Genético.

2.3 NOVELTY SEARCH

O Novelty Search surgiu da ideia de resolver a desconexão entre aprendizado de máquina e vida artificial (11). Mais especificamente, o aprendizado de máquinas foca em vencer cenários enganosos e superar ótimos locais e a vida artificial em criar cenários abertos com crescimento contínuo. Porém, é argumentado que o conceito de Novelty Search resolve os dois problemas através da exploração de soluções inovadoras sem explicitar um objetivo específico. Além disso, é argumentado que quanto mais ambicioso o objetivo, maior a possibilidade da busca ser enganada por ótimos locais, já que, a busca orientada a um objetivo (fitness) não necessariamente recompensa as iterações em direção ao objetivo da otimização.

Esse problema com as funções objetivos são chamadas stepping stones (11), onde esse problema acontece quando a função objetivo utilizada em algoritmos de otimização não é capaz de capturar adequadamente as soluções que podem levar a um desempenho melhor no longo prazo. Um dos exemplos é o encontrado na movimentação bípede em (8), em que a função objetivo recompensa os indivíduos que permanecem longe do chão. Dessa forma, indivíduos que mantém as pernas estendidas são recompensados. Porém, a movimentação de tais indivíduos a curto prazo pareça mais benéfica, a longo prazo possuem uma movimentação rígida e não natural. Em contrapartida, a abordagem baseada em novidade pode permitir que o algoritmo explore diferentes movimentos, incluindo dobrar a perna inicialmente, o que pode parecer pior em termos de fitness inicial, por se aproximar do chão, mas no longo prazo percorre uma distância maior com uma movimentação mais fluída e natural.

Outro exemplo ocorre na navegação em labirintos (11), em que o uso de uma função objetivo leva a recompensar exclusivamente soluções que se aproximem do objetivo, o que pode levar a becos sem saída e descartar indivíduos que encontrariam maneiras de sair do labirinto. Já para o caso da otimização pela novidade, o indivíduo é recompensado por bater em uma parede diferente das demais já batidas. O argumento principal é que, ao bater repetidamente em várias paredes, essa ação de bater em paredes deixa de ser uma novidade, e o algoritmo passa a recompensar indivíduos que vão mais longe e com o passar do tempo, os indivíduos chegam ao final do labirinto, mesmo que não seja explicitamente indicado que esse é o objetivo.

Ainda é discutida a natureza bio-inspirada da novidade. Embora não exista evidência de que a natureza faça uma busca explícita por novidades, há evidências de que quando a pressão da seleção é reduzida, a inovação ganha força (8). A ideia é que, as mutações aleatórias ocorridas

na natureza são duramente pressionadas pela seleção natural, preservando as características adaptativas. Porém, existem mecanismos na natureza que permitem a novidade quando o indivíduo atende requisitos mínimos. Uma forma de enxergar isso, é um indivíduo com comportamento diferente dos demais, que ao explorar uma fonte de recursos que outros não exploram, consegue vantagem sobre os demais. Além disso, existem evidências de buscas ativas pela novidade, onde indivíduos com fenótipos raros são mais escolhidos em relação aos com fenótipos comuns, dessa forma, favorecendo diretamente a novidade. Levando em conta uma baixa pressão de seleção, indivíduos inovadores tem mais espaço para sobreviver, o Novelty Search pode ser visto como uma forma de acelerar esse processo, uma vez que, indivíduos inovadores em relação aos seus vizinhos são recompensados, impulsionando o avanço em direção a evolução.

Lehman (10), propõe uma discussão sobre o real impacto da pressão seletiva sobre a evolução ao apresentar duas explicações, por meio de experimentos, para a evolução biológica sem a pressão de um indivíduo se adaptar. Sabendo que a evolução não adaptativa se refere a capacidade de um organismo de gerar variabilidade genética e evoluir, aumentando a capacidade de evolução mesmo na ausência de pressões adaptativas específicas se desviando do tradicionalmente conhecido, onde a evolução é vista como um processo em que os organismos se adaptam ao ambiente por meio da seleção natural. Aliando esse argumento ao problema de funções objetivo ambiciosas (stepping stones), é possível ver grandes benefícios de se utilizar a busca pela novidade para a otimização de objetivos.

Os algoritmos tradicionais baseados em população buscam otimizar o custo para o problema aplicado recompensando indivíduos que tenham as melhores soluções encontradas até o momento. Porém, tal métrica pode direcionar o algoritmo a convergências muito antecipadas e estagnar em ótimos locais ao descartar possíveis soluções ou comportamentos que a primeiro momento possuem uma depreciação do objetivo, mas que no futuro podem se tornar soluções melhores. Já na busca no espaço comportamental, tal problema não existe, pois indivíduos com comportamentos diferentes são recompensados.

2.3.1 Definição

O objetivo principal do NS é aumentar a diversidade de soluções em algoritmos baseados em população. Essa diversidade é alcançada através da troca de buscas no espaço convencional pela busca no espaço comportamental. Em geral, indivíduos de uma mesma população tendem a convergir para o mesmo ponto no espaço de otimização de objetivos. Porém, no espaço comportamental tal tendência não é observada, uma vez que, indivíduos com comportamentos diferentes são recompensados em detrimento do objetivo de otimizar o custo, trazendo assim uma diversidade maior em relação aos algoritmos tradicionais.

Os comportamentos recompensados, chamados novidade, são medidos pela a distância Euclidiana do indivíduo com sua vizinhança. Essa vizinhança é o que define o histórico de comportamentos de indivíduos analisados no passado e o quão diferente é uma solução. Sendo assim, com o passar das gerações, um indivíduo com um comportamento considerado inovador em um ponto, pode deixar de ser inovador conforme mais soluções explorem aquele espaço. Basicamente, o espaço comportamental pode ser traduzido como densidade de clusters, em que soluções habitantes de clusters mais esparsos possuem maior novidade da mesma forma que áreas com menos clusters possuem soluções com maior novidade.

Então, a novidade de um indivíduo pode ser calculada pela distância média para seus k-próximos vizinhos. Com isso temos que se um determinado ponto possui uma distância pequena, este ponto está em uma área muito povoada, logo, possui uma novidade pequena. Dessa forma, basta calcular a esparsidade de um indivíduo no espaço. Para isso, um indivíduo x possui sua esparsidade p calculada pela formula 2.3:

$$p(x) = \frac{1}{k} \sum_{i=1}^{k} d(x, x_i) \tag{2.3}$$

onde p é a esparsidade de um indivíduo $x \in S$ e S o conjunto de candidatos para medir a novidade, k é o número de vizinhos escolhidos do conjunto $N = (x_1, x_2, ..., x_k) \subseteq S$ e $d(x_i, x_j)$ é a métrica de distância de $x_i, x_j \in N$.

Levando em conta que a métrica de distância é diretamente dependente do domínio, a métrica e o tamanho da vizinhança devem ser adaptadas para a aplicação. Mais especificamente, para um problema de otimização contínua, a distância Euclideana ou Manhattan pode ser aplicada. A distância Euclideana, derivada do teorema de Pitágoras, define a distância como o comprimento de uma linha reta entre dois pontos em R^n. Sendo u e v, com $|u| = |v| = n$, dois vetores 1-D, e w, com $|w| = n$, vetor de pesos de cada coordenada, a distância Euclideana de u e v tem seu calculo é definido por (23):

$$Ed(u, v) = (\sum_{i=0}^{n} (w_i \times (|u_i - v_i|)^2))^{\frac{1}{2}} \forall 0 \leq i \leq n - 1 \tag{2.4}$$

Já para problemas discretos, como problemas de roteamento, em que a solução é dada por restrições e permutações e a similaridade entre soluções é necessária, a distância Hamming pode ser utilizada. A distância Hamming é definida (23) pela proporção de dissimilaridade entre dois vetores 1-D u e v, com $|u| = |v| = n$. Então, é medida a diferença de cada elemento i de u e v não correspondentes, seguindo a formula 2.5:

$$Hd(u, v) = \sum_{i=0}^{n} (u_i \neq v_i) \forall 0 \leq i \leq n - 1 \tag{2.5}$$

2.4 CONSIDERAÇÕES FINAIS

Em suma, o TSP é um problema importante e desafiador que tem sido estudado há muitas décadas. Sua resolução é crucial em diversas áreas práticas, além de ser um objeto de estudo interessante em teoria da complexidade computacional e matemática discreta. As soluções para o TSP são diversas e evoluem constantemente, com o desenvolvimento de novas técnicas e algoritmos. A busca por soluções cada vez mais eficientes e precisas para o TSP é um desafio constante e uma oportunidade de aprendizado e avanço para a ciência da computação.

Nesse capítulo foram apresentadas duas abordagens: o Algoritmo Genético e o Novelty Search. Para ambas aplicações, foram levantadas suas características, suas naturezas bio-inspiradas e suas meta-heurísticas. Além disso, foi levantado também para o Novelty Search as possíveis melhorias que podem ser obtidas ao utilizar a busca no espaço comportamental em conjunto com a otimização de objetivos. A seguir serão apresentados trabalhos relacionados que utilizam o espaço comportamental para trazer variabilidade a algoritmos baseados em população.

3 TRABALHOS RELACIONADOS

Neste capítulo, será realizada uma revisão dos trabalhos mais relevantes que abordam o uso de algoritmos genéticos, Problema do Caixeiro-Viajante (TSP) e busca por novidade. Essas áreas têm despertado grande interesse na comunidade científica devido à sua aplicabilidade em problemas complexos de otimização.

3.1 ALGORITMOS GENÉTICOS

Diversos estudos têm sido conduzidos para comparar os resultados da utilização de algoritmos genéticos com variações e melhorias no contexto do Problema do Caixeiro-Viajante (TSP). Um exemplo relevante é o estudo de Whitley et al. (1992), intitulado "The Traveling Salesman and Sequence Matching: A Comparison of Genetic Sequencing Operators"(21). Nessa pesquisa, os autores propõem e comparam diferentes operadores genéticos, como o OX (Order Crossover), CX (Cycle Crossover) e PMX (Partially Mapped Crossover), juntamente com diferentes estratégias de seleção. O objetivo é identificar os operadores e estratégias mais eficazes para resolver o TSP. Além desse estudo, várias outras pesquisas realizam comparações e propõem abordagens inovadoras. Por exemplo, o estudo de Mühlenbein et al. intitulado "Parallel Genetic Algorithms, Population Genetics and Neutrality"aborda a utilização de algoritmos genéticos paralelos para o TSP, explorando a relação entre a população genética, a convergência do algoritmo e a preservação da diversidade. Além disso, surveys como "Literature Survey On Travelling Salesman Problem Using Genetic Algorithms"por Anitha Rao e IISandeep Kumar Hegde (17), fornecem uma visão abrangente das diferentes propostas e comparações entre operadores utilizados em algoritmos genéticos aplicados ao TSP. Esses estudos demonstram a ampla gama de abordagens e técnicas disponíveis para melhorar a eficácia dos algoritmos genéticos no contexto do TSP.

Apesar de melhorias nas abordagens utilizando algoritmos genéticos, ainda existem desafios como convergência prematura, falta de diversidade e problemas de escalabilidades. Esses desafios motivam os pesquisadores a buscar a integração de novas técnicas, como a busca por novidade para sobrepor as limitações do algoritmo genético tradicional.

3.2 NOVELTY SEARCH PARA O TSP

A busca por novidades tem sido amplamente explorada em diversos problemas de otimização. No contexto do Problema do Caixeiro-Viajante (TSP), a busca por novidade tem demonstrado resultados promissores ao melhorar a diversidade de soluções e evitar ótimos locais. Ao promover a exploração e incentivar a busca por soluções não convencionais, a busca por novidade complementa a capacidade dos algoritmos em explorar o espaço de busca e oferece uma solução potencial para os desafios enfrentados na resolução de problemas complexos de otimização, como o TSP.

No artigo intitulado "Traveling Salesman Problem: a Prospective Review of Recent Research and New Results with Bio-Inspired Metaheuristics and Novelty Search", de Eneko Osaba, Xin-She Yang e Javier Del Ser (16), os autores investigam a aplicação dos algoritmos bioinspirados Firefly, Particle Swarm Optimization (PSO) e Bat Algorithm (BA) para resolver o problema do TSP. Além disso, eles realizam uma comparação entre as versões tradicionais desses algoritmos e suas variantes que incorporam a busca por novidade, visando aprimorar a

diversidade de soluções e evitar a convergência prematura para ótimos locais. Os resultados obtidos revelam benefícios significativos no uso da busca por novidade, comprovando melhorias na diversidade das soluções encontradas e fornecendo evidências promissoras para a aplicação dessa abordagem integrada no contexto do TSP.

Embora estudos anteriores tenham explorado o uso de Algoritmos Genéticos para abordar o Problema do Caixeiro-Viajante (TSP) e investigado os benefícios da Busca por Novidade em problemas de otimização, há uma lacuna na pesquisa que se concentra na integração de Algoritmos Genéticos, TSP e Busca por Novidade. O presente estudo tem como objetivo preencher essa lacuna, investigando a eficácia da incorporação da Busca por Novidade nos Algoritmos Genéticos para resolver o TSP. Ao combinar os pontos fortes dos Algoritmos Genéticos na exploração do espaço de soluções e a Busca por Novidade na promoção da exploração e diversidade, espera-se que essa abordagem integrada gere soluções aprimoradas para o TSP. Além disso, os resultados obtidos serão comparados com os experimentos conduzidos por Eneko Osaba, Xin-She Yang e Javier Del Ser, visando avaliar a melhoria alcançada neste estudo.

4 MODELAGEM

4.1 MODELAGEM DO ALGORITMO GENÉTICO AO TSP

Como o definido no capítulo 2, o TSP consiste em encontrar o menor Ciclo Hamiltoniano no conjunto de cidades. Um conjunto de cidades é um grafo, onde cada vértice é uma cidade e as arestas são os custos de viajar de uma cidade a outra. O custo do ciclo é definido como a soma de todas as arestas escolhidas. Além disso, o conjunto de cidades é definido como um grafo completo.

A Aplicação ao problema segue as etapas definidas na figura 2.1, sendo elas, Codificação, Função Fitness, Seleção, Crossover, Mutação e Condição de finalização. Como essas etapas definem o fluxo que o algoritmo segue, na modelagem o primeiro passo é a codificação do algoritmo. A codificação, nesse caso, consiste da definição de um cromossomo como uma permutação de cidades, definindo uma rota por todas as cidades, onde nenhuma cidade pode ser repetida. Cada gene do cromossomo representa uma cidade do grafo. E o conjunto de cromossomos forma uma população. Analogamente, a codificação de um cromossomo para o TSP consiste em formar permutações de cidades de uma instância formando um caminho percorrido pelo vendedor, logo, formam uma solução do problema. Sendo assim, com uma instância do TSP contendo n cidades, um cromossomo deverá ter uma sequência de n cidades e este cromossomo uma solução para o problema.

O segundo passo é a definição da função Fitness. Como para o caso do TSP é desejável encontrar o menor caminho em um conjunto de vértices, se uma solução x possui um caminho maior que uma solução y, a solução x terá um fitness menor que a solução y. Nesse caso, a avaliação do fitness do indivíduo é dada puramente pelo somatório das distâncias da permutação de cidades. Mais especificamente, para uma solução X é realizado um laço da posição $i = 0$ até a posição $n - 1$, onde em cada iteração é acumulada a distância de $d(X_i, X_{i+1})$. Como ao final da expedição, deve-se retornar ao ponto inicial, é somado também a distância $d(X_n, X_0)$. Portanto, uma permutação que possui menor distância, possuindo um fitness maior, é mais recompensada em relação a uma permutação com maior distância.

O terceiro passo, a etapa seleção, consiste na aplicação do método de torneios, que basicamente é realizado n torneios, onde cada torneio coloca k indivíduos selecionados aleatoriamente para competirem entre si. Tal competição é definida pelo fitness de cada indivíduo, onde aquele com o maior fitness, nesse caso a menor distância, é o vencedor. O vencedor será incluído na próxima geração e possui a chance de reproduzir e passar suas características para seus herdeiros. Além disso, é admitido também um sistema de elitização, onde a melhor solução já encontrada é sempre recolocada na população no lugar da pior solução corrente. Porém, esse sistema pode impactar de forma que as solução fiquem presas em um ótimo local, mas com o benefício de preservar sempre o melhor fitness encontrado.

Partindo para a parte de operadores genéticos, as restrições do problema devem ser consideradas. As restrições são: o problema consiste de um grafo completo; a não repetição de cidades; e todas as cidades devem estar presentes no indivíduo resultante da função. Sendo o crossover uma troca entre cromossomos, bastaria selecionar um ponto de corte l e particionar um parente P_1 e um parente P_2, ambos de tamanho n. Após isso, criar filhos F_1 e F_2 que receberão respectivamente $F_1 = P_1[0 : l] + P_2[l : n]$ e $F_2 = P_2[0 : l] + P_1[l : n]$. Sendo assim, dados $P_1 = [1, 2, 3, 4, 5, 6]$ e $P2 = [3, 2, 1, 6, 5, 4]$, e o ponto de corte $l = 3$ a aplicação da função crossover resultaria em dois filhos $F_1 = [1, 2, 3, 6, 5, 4]$ e $F_2 = [6, 5, 4, 1, 2, 3]$. Porém, isso não

funcionaria para todos os casos, visto que se $P_3 = [1, 2, 3, 4, 5, 6]$ e $P_4 = [6, 5, 4, 3, 2, 1]$, duas permutações totalmente válidas, e o ponto de corte $l = 3$, resultaria em filhos que violam a restrição de não repetição de cidades e a restrição de todas as cidades estarem presentes. Por isso, a função de crossover deve ser elaborada pensando nas restrições.

Para solucionar o problema é aplicado o seguinte método, selecionar dois pontos, onde a porção interior que tais pontos formam no primeiro parente são diretamente herdados nas mesmas posições pelo primeiro filho, e as cidades que não estão no primeiro filho inseridas nas posições vazias na ordem em que aparecem no segundo parente. Analogamente, o mesmo deve ser realizado para o segundo parente e para o segundo filho. Sendo assim, o processo de crossover é aplicado nos seguintes passos:

1. Selecionar dois pontos i e j, onde $0 \leq j < n$;
2. Copiar posições i até j de P_3 e inserir nas mesmas posições de F_3;
3. Copiar posições i até j de P_4 e inserir nas mesmas posições de F_4;
4. Inserir cidades faltantes em F_3 na ordem que aparecem em P_4; e
5. Inserir cidades faltantes em F_4 na ordem que aparecem em P_3.

Com isso, para os parentes $P_3 = [1, 2, 3, 4, 5, 6]$ e $P_4 = [6, 5, 4, 3, 2, 1]$ e $i = 1$, e $j = 3$, a aplicação dos passos 2 e 3 resultam em $F_3 = [, 2, 3, 4, ,]$ e $F_4 = [, 5, 4, 3, ,]$. Seguindo os passos 4 e 5, os filhos resultantes são $F_3 = [6, 1, 3, 4, 5, 2]$ e $F_4 = [1, 5, 4, 3, 2, 6]$.

Ainda na parte de operadores genéticos e considerando que um indivíduo possuirá todas as cidades possíveis em seus genes, a mutação acontecerá por meio da troca da ordem de dois genes. Sendo assim, são selecionadas duas posições aleatórias i e j, e tais posições aleatórias selecionadas são trocadas entre si. Então, para o indivíduo $I = [1, 2, 3, 4, 5, 6]$, $i = 1$ e $j = 4$, o indivíduo resultante é $I' = [1, 5, 3, 4, 2, 6]$.

Lembrando que, para ambos os casos das funções de Mutação e Crossover existe uma probabilidade de se aplicar as funções. Para isso, é dado dois valores de corte, taxa de mutação M_t e taxa de crossover C_t e ao iniciar ambas as funções, um número aleatório é sorteado e se este número for menor que a respectiva taxa de corte, a função é aplicada nos indivíduos passados para as funções.

Para finalizar, é necessário estabelecer um critério de parada. O critério mais básico é um número limite de gerações que podem ser executadas. Sendo assim, após o limite de gerações, o algoritmo para e devolve o melhor resultado encontrado e outros dados para análise. Além disso, podem ser aplicados outros métodos, como a medida de estagnação do algoritmo, a entropia, entre outros. Para essa aplicação, por se tratar de uma medida comparativa, apenas o limite de gerações é utilizado como critério de parada.

4.1.1 Implementação GA

A implementação do algoritmo genético consiste em utilizar o esquema proposto em (22) com a inicialização de populações com o método de busca gulosa (12) e método de seleção de parentes por torneio pela sua eficiência superior a outros métodos de seleção (3) e eficiência superior no TSP (18).

A implementação do algoritmo base pode ser vista no Algoritmo 1. Basicamente uma população inicial e sua fitness é estabelecida a partir de uma busca gulosa com uma melhor solução é selecionada inicialmente. A partir disso, o laço principal é iniciado até que aconteçam

$generationsNumber$ de gerações. No laço principal, são realizadas as seleções e evoluções dos indivíduos formando uma nova população (geração). Em cada iteração, uma nova $bestSolution$ da geração é selecionada. Se essa $bestSolution$ for melhor que a $globalBestSolution$, ela se torna a melhor solução global. A $globalBestSolution$ é reinserida no lugar da pior solução ao final de cada geração. Ao atingir o critério de parada, a permutação com menor distância total é devolvida.

Algoritmo 1 Algoritmo Genético:

```
1: population ← (greedyPopulation(), calculateFitness()) // Ref Alg 6 e 5
2: globalBestSolution ← min(population)
3: for i ← 0; i < generationsNumber; i ← i + 1 do
4:     population ← (evolve(), calculateFitness()) // Ref Alg 2 e 5
5:     bestSolution ← min(population)
6:     worstSolution ← max(population)
7:     if bestSolution > globalBestSolution then
8:         globalBestSolution ← bestSolution
9:     end if
10:    population[worstSolution] ← globalBestSolution
11: end for
12: return globalBestSolution
```

A cada geração, os indivíduos passam por um processo de reprodução e evolução. Porém não é garantido que cada indivíduo da população se reproduza, sofra mutação ou ambos. Esses processos acontecem ao rodar um número aleatório entre 0 e 1 a partir de uma distribuição discreta uniforme (4). O algoritmo para a evolução é dado no algoritmo 2. Nesse estágio, os indivíduos selecionados pelo método de torneios são enviados para realizar o crossover em pares. Cada crossover retorna um par, que podem ser novos indivíduos ou os mesmos enviados. Depois disso, os indivíduos são enviados para a mutação e inseridos na nova população. Por fim, a nova população formada é retornada. Vale ressaltar que existe a possibilidade da população de entrada ser exatamente igual a população de saída, porém, essa possibilidade é baixa.

Algoritmo 2 Evolução:

```
1: n ← size(population)
2: newPopulation ← []
3: population ← Tournament() // Ref Alg 7
4: for i ← 0; i < n; i ← i + 2 do
5:     offspring1, offspring2 ← Crossover(population[i], population[i + 1]) // Ref Alg 3
6:     newPopulation[i] = Mutate(offspring1) // Ref Alg 4
7:     newPopulation[i + 1] = Mutate(offspring2) // Ref Alg 4
8: end for
9: return newPopulation
```

No algoritmo de Evolução (Algoritmo 2) são chamadas duas funções, Crossover (Algoritmo 3) e Mutate (Algoritmo 4). Como uma solução se trata de uma permutação, não pode haver nenhuma repetição de cidades em uma solução. Além disso, todas as cidades devem estar presentes no array. O disparo da função de crossover acontece a partir de selecionar um número aleatório entre 0 e 1, se o número for menor que a taxa de crossver C_t (crossoverRate), é realizado o crossover entre os parentes passados para a função, caso contrário, não são gerados novos indivíduos e os cromossomos originais são retornados. Com isso, a função

de crossover representada no Algoritmo 3 segue uma ideia simples: dadas duas permutações X e Y de tamanho n, selecionar dois pontos aleatórios (i, j), com $0 \leq i \leq j < n$. Formar dois subconjuntos $x' \subseteq X$ e $y' \subseteq Y$, de tamanho $k = j - i + 1$, onde $x' = [X_i, X_{i+1}, ..., X_j]$ e $y' = [Y_i, Y_{i+1}, ..., Y_j]$. Com isso, formar dois novos subconjuntos u e v de tamanho $l = n - k$, onde $u = \{a | a \in Y, a \notin x'\}$ e $v = \{a | a \in X, a \notin y'\}$, recebendo os valores de X e Y na ordem em que estão nos conjuntos. A partir desse ponto, formar dois novos conjuntos A e B, onde $A = [u_1, u_2, ..., u_{i-1}, x'_1, x'_2, ..., x'_k, u_i, u_{i+1}, ..., u_l]$ e $B = [v_1, v_2, ..., v_{i-1}, y'_1, y'_2, ..., y'_k, v_i, v_{i+1}, ..., v_l]$.

Algoritmo 3 Crossover:

```
1: if Rand.float(0, 1) < crossoverRate then
2:     start ← Rand.int(0, numCities)
3:     end ← Rand.int(start, numCities)
4:     offspring1[start : end] ← parent1[start : end]
5:     offspring2[start : end] ← parent2[start : end]
6:     remaining1 ← {a|a ∈ parent2, a ∉ offspring1}
7:     remaining2 ← {a|a ∈ parent1, a ∉ offspring2}
8:     idx1, idx2 ← 0, 0
9:     for i ← 0; i < numCities; i ← i + 1 do
10:        if offspring1[i] == NULL then
11:            offspring1[i] ← remaining1[idx1]
12:            idx1 ← idx1 + 1
13:        end if
14:        if offspring2[i] == NULL then
15:            offspring2[i] ← remaining2[idx2]
16:            idx2 ← idx2 + 1
17:        end if
18:    end for
19: else
20:    offspring1, offspring2 ← parent1, parent2
21: end if
22: return offspring1, offspring2
```

Para o caso do algoritmo de Mutação (Algoritmo 4), a restrição é a mesma que para o Crossover (Algoritmo 3), não pode repetir cidades e todas as cidades devem existir no conjunto resultante. Assim como para o crossover, é sorteado um número entre 0 e 1 e se o valor for menor que a taxa de mutação M_t (mutationRate) a mutação para o indivíduo selecionado acontece. Sabendo que a mutação modifica um gene de um cromossomo (indivíduo), e levando em conta a restrição, a mutação acontece da seguinte maneira: Dado um conjunto X de tamanho n, selecionar dois inteiros aleatórios i e j, onde $0 \leq i, j < n$. Com isso, formar uma nova permutação X' de tamanho n, onde a X' é X as posições i e j de X trocadas, resultando em $X' = [X_1, X_2, ..., X_{i-1}, X_j, X_{i+1}..., X_{j-1}, X_i, X_{j+1}, ..., X_n]$.

Algoritmo 4 Mutação:

```
1: mutation ← solution
2: if Rand.float() < mutationRate then
3:     idx1 ← Rand.int(0, numCities)
4:     idx2 ← Rand.int(0, numCities)
5:     mutation[idx1] ← solution[idx2]
6:     mutation[idx2] ← solution[idx1]
7: end if
8: return mutation
```

Ainda no algoritmo 1, é referenciada a função Fitness (Algoritmo 5). Partindo do ponto de que é utilizado o fitness de um indivíduo como o somatório das distâncias entre as cidades, e o foco do algoritmo é em minimizar esse valor. Basicamente, a função fitness implementada consiste em somar o valor da distância entre um par de cidades c_1 e c_2 a partir da matriz de distâncias da instância do problema executado, repetindo para toda a sequência do cromossomo e somando a distância do último gene com o primeiro. Dado um conjunto X de tamanho n, e dado a matriz de distâncias M_d de tamanho $n \times n$. O fitness do indivíduo representado pelo conjunto X é $d(X) = \sum_{i=0}^{n-1} M_d[X_i][X_{i+1}] + M_d[n][0]$.

Algoritmo 5 Fitness:

```
1: for all solution ∈ population do
2:     distance ← 0
3:     for i ← 0; i < size(solution); i ← i + 1 do
4:         city1 ← solution[i] + 1
5:         city2 = solution[(i + 1) mod size(solution)] + 1
6:         distance ← distance + distMatrix[city1][city2]
7:     end for
8:     fitness[solution] = distance
9: end for
```

A opção de inicialização pode ser obtida de forma heurística ou aleatória (24). No artigo (12), é empregado a utilização de uma busca gulosa como inicialização das populações do Algoritmo Genético aplicado ao p-median problem, obtendo resultados 4% melhores para a população inicial. Sendo assim, a inicialização da implementação utilizada foi adotado a busca gulosa para inicializar o algoritmo. A busca gulosa (2) consiste do Algoritmo 6:

Algoritmo 6 Busca Gulosa TSP:

```
1: for all pop ∈ P_i do
2:     pop[0] ← random(cities)
3:     unvisited ← x|x ∈ cities, x ≠ pop[0]
4:     i ← 1
5:     while !empty(unvisited) do
6:         prevCity ← pop[i − 1]
7:         nextCity ← FindNear(prevCity, unvisited)   // "Encontra a cidade mais proxima de prevCity
           em unvisited"
8:         unvisited ← x|x ∈ cities, x ∉ pop
9:         pop[i] ← nextCity
10:        i ← i + 1
11:    end while
12: end for
```

A seleção por torneio baseia-se em rodar n, sendo n o tamanho da população. Em cada torneio, k indivíduos (TOURNAMENT_SIZE) são selecionados aleatoriamente e competem entre si com um vitorioso no fim. O vencedor é selecionado pelo fitness da solução, sendo assim, o indivíduo com melhor fitness é o vencedor e será incluído para a próxima geração. O método de torneios da chances a todos os indivíduos da população de reproduzirem e balanceando a diversidade em detrimento da velocidade de convergência, o que não é necessariamente ruim, visto que, uma convergência muito antecipada diminui a qualidade do resultado final. Comumente, o número utilizado de indivíduos competindo são dois (chamado torneio binário), nesta implementação, é utilizado 5 indivíduos obtidos experimentalmente como o ótimo. O algoritmo 7 é o responsável por implementar os torneios.

Algoritmo 7 Seleção por torneio:

```
1: n ← size(population)
2: winners ← []
3: for i ← 0; i < n; i ← i + 1 do
4:     tournament ← []
5:     for j ← 0; j < TOURNAMENT_SIZE; j ← j + 1 do
6:         tournament[j] ← random(population)
7:     end for
8:     winners[i] ← min(tournament)
9: end for
10: population ← winners
```

Vale ressaltar que, quanto maior o tamanho do torneio, menor a capacidade de um indivíduo contribuir para a diversidade da população. Sendo que, um torneio maior, maior a probabilidade de indivíduos de maior fitness sobrepujar indivíduos de menor fitness. Com isso, temos dois casos podem acontecer para um indivíduo não ser selecionado: 1. O indivíduo em questão não entrou em nenhum torneio. 2. O indivíduo perdeu os torneios em que entrou.

4.2 APLICAÇÃO DO NOVELTY SEARCH AO TSP

Embora em algumas aplicações, como (8), é defendida a utilização da novidade sem qualquer uso de funções de custo, a busca puramente pelo espaço comportamental no TSP traz alguns problemas fundamentais devido a diferença conceitual do problema. Esse problema acontece pelo fato do TSP possuir o objetivo de obter uma permutação ótima, ao contrário de problemas de robótica onde o interesse é em otimizar comportamentos. Por isso, para essa pesquisa é utilizado o NS em conjunto com o GA tradicional, nomeado GA_{NS}, formando uma mescla entre busca por objetivo, orientando a evolução da população, e a busca no espaço comportamental, trazendo maior encaixe com a natureza bio-inspirada ao introduzir comportamentos diferentes nas gerações.

Originalmente, os indivíduos que têm seus comportamentos avaliados competem com seus vizinhos. Porém, o número de vizinhos e a vizinhança é dependente do problema e deve ser ajustada dependendo do problema e da abordagem. Para esse estudo, a avaliação individual é utilizada 1 para n, com n sendo todos os indivíduos elegíveis para a novidade.

Os indivíduos elegíveis para a avaliação são os indivíduos da população principal (A) e os indivíduos descartados na última geração. Esses indivíduos descartados formam uma população reserva (B) sem repetições de indivíduos. Com isso, temos que os indivíduos descartados ainda possuem chance de retornarem para a próxima geração, baseado na novidade individual.

Levando em conta que deseja-se trazer maior variabilidade nas reproduções, porém mantendo a direção em que o algoritmo caminha, é aplicada uma análise para o momento correto de introduzir os indivíduos inovadores na população. Essa análise é dada pela estagnação do fitness do melhor indivíduo encontrado. Isso significa que, quando o melhor fitness global não melhora durante um certo número de gerações, k indivíduos com maior taxa de inovação são inseridos na população, que passará pela seleção posteriormente. Além disso, quando o mecanismo de novidade é disparado diversas vezes sem a melhora do fitness, a taxa de ativação é reduzida, aumentando o número de ativações e uma população gerada aleatoriamente é inserida em (B) para a avaliação de novidade.

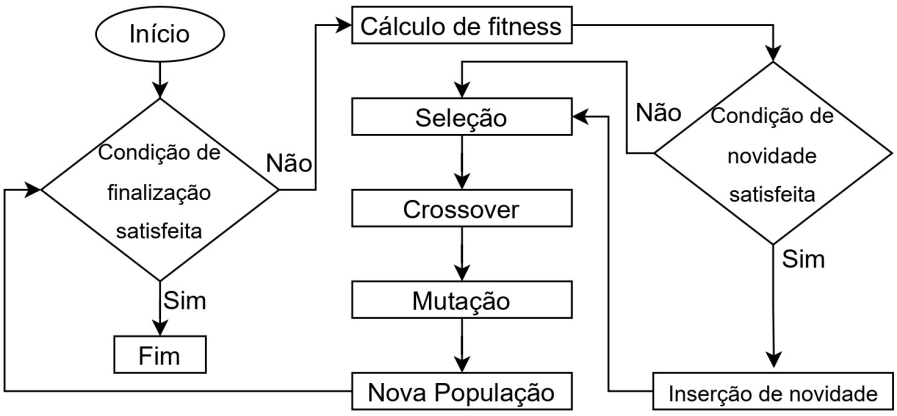


Figura 4.1: Fluxograma com etapas realizadas pelo Novelty Search.

Na figura 2.1 é possível verificar que o funcionamento do algoritmo é basicamente o GA tradicional, porém com a adição do mecanismo de novidade. Quando a condição de disparo é satisfeita, os indivíduos mais inovadores são inseridos no lugar de indivíduos aleatórios. Caso contrário, a seleção ocorre normalmente. Tendo em vista a modelagem do NS para o GA, pode-se constatar que todo o algoritmo segue normalmente, porém, a variabilidade nas populações é inserida, atingindo o objetivo proposto, maior exploração no espaço de busca.

4.2.1 Implementação NS

Foi adotado para essa pesquisa o modelo de NS apresentado anteriormente com o GA. Basicamente, o funcionamento do GA é o tradicional mas com interferências nas populações feitas pelo NS. Embora o autor do artigo (16) utilize o disparo do mecanismo de novidade baseado em um sorteio de um número aleatório com probabilidade de escolha por uma distribuição normal, nessa pesquisa o modelo é adaptado para disparar com base na estagnação da fitness entre as populações.

A cada x evoluções das populações sem que haja melhora na fitness do melhor indivíduo já encontrado, o mecanismo é disparado e os m indivíduos da população B com maior novidade são inseridos na população A. Vale notar que não existe um consenso do número de indivíduos que devem ser reinseridos na população principal. Outra diferença é que os indivíduos selecionados pelo mecanismo substituem indivíduos aleatórios e não os m piores.

Levando em conta que o mecanismo de disparo da novidade parte da estagnação do GA, é possível que mesmo com a novidade sendo inserida na população principal o algoritmo continue estagnado. Por isso, o limite x varia conforme as gerações. Essa variação acontece com $x \leftarrow x/2 + 1$ ao atingir $2 \times x$ sem melhoria no fitness, sendo 2 o menor valor de x, e $x \leftarrow 100$ ao melhorar o fitness.

Sabendo que a dinâmica do limite x é atingida ao não encontrar melhorias no fitness, uma população aleatória é adicionada a população B. Essa população aleatória adicionada a B compete com os demais indivíduos para a avaliação da novidade e possuem chance de entrar na população principal A e se reproduzir.

O algoritmo de busca de novidade, Novelty Search (Algoritmo 8), baseia-se numa ideia simples, o indivíduo que possui a maior distância em relação a todos os demais, é o indivíduo mais inovador. Partindo do princípio que cada indivíduo é uma permutação de n cidades, a distância entre um par de indivíduos é medida através da distância Hamming 2.5. Dado um conjunto de cidades C, de tamanho n. E dado uma população P, um conjunto de permutações de C, de tamanho k. Selecionar um indivíduo $I \in P$, a novidade de I é calculada por $N(I) = \frac{1}{k} \sum_{i=0}^{k} D_h(I, P_i) \forall P_i \subseteq C$. O processo é repetido para todos os itens em P. Ao final, remover de P e retornar o indivíduo com maior média de distância D_h.

Algoritmo 8 Novelty Search:

```
1: n ← size(sparePopulation)
2: distances ← []
3: for i ← 0; i < n; i ← i + 1 do
4:     for j ← 0; j < n; j ← j + 1 do
5:         dist ← dist + hammingDistance(sparePopulation[i], sparePopulation[j])
6:     end for
7:     distances[i] ← dist/n
8: end for
9: bestNovelty ← max(distances)
10: sparePopulation ← sparePopulation − bestNovelty
11: return bestNovelty
```

O algoritmo 8 define o mecanismo para encontrar o indivíduo com a maior média de distância Hamming para os demais indivíduos, consequentemente, encontra o indivíduo com maior novidade. Porém, tal mecanismo precisa ser disparado. Para isso, a implementação do algoritmo 9 segue o esquema da figura 4.1, consistindo da inserção da verificação da estagnação do fitness do melhor indivíduo global como condição de disparo. Essa estagnação é dada por uma variável $limit$, que ao atingir o valor $x \leftarrow ACTIVATION\_LIMIT$, inicia o processo de inserção de novidade. Quando $limit$ atinge $2 \times x$, o limite dinâmico é ativado modificado x com $x \leftarrow x/2 + 1$, aumentando dessa forma a quantidade de vezes em que o mecanismo é disparado. Além disso, quando o limite dinâmico é ativado, uma população aleatória é inserida na população reserva, que competirá com os demais para medir suas respectivas novidades. Por fim, cada indivíduo que retorna do algoritmo Novelty Search (Algoritmo 8), substitui um indivíduo aleatório na população principal. Quando uma melhora no fitness é encontrada, tanto o $limit$ quanto x tem seus valores iniciais reestabelecidos.

Algoritmo 9 Novelty GA:

```
1: population ← greedyPopulation() // Ref Alg 6
2: populationSize ← size(population)
3: globalBestSolution ← min(population)
4: limit ← 0
5: x ← ACTIVATION_LIMIT
6: for i ← 0; i < generationsNumber; i ← i + 1 do
7:     limit ← limit + 1
8:     if limit mod x == 0 then
9:         if limit == 2 × x then
10:            x ← x/2 + 1
11:            sparePopulation ← sparePopulation + RandomPopulation()  // "Adiciona uma
               população aleatória"
12:        end if
13:        for k ← 0; k < REPLACE_SIZE; k ← k + 1 do
14:            randNumber ← Rand.int(0, populationSize)  // "Gera um inteiro entre 0 e population-
               Size"
15:            population[randNumber] ← noveltySearch(sparePopulation) // Ref Alg 8
16:        end for
17:    end if
18:    sparePopulation ← population
19:    population ← Evolve(), Fitness() // Ref Alg 2 e 5
20:    sparePopulation ← sparePopulation + population
21:    bestSolution ← min(population)
22:    worstSolution ← max(population)
23:    if bestSolution > globalBestSolution then
24:        globalBestSolution ← bestSolution
25:        limit ← 0
26:        x ← ACTIVATION_LIMIT
27:    end if
28:    population[worstSolution] ← globalBestSolution
29: end for
```

Ainda no algoritmo 9, a formação da população reserva segue um modelo simples, *sparePopulation* guarda a população principal (*population*). A população principal, passa pelo processo de seleção e evolução, modificando seus indivíduos. Com isso, essa nova população é adicionada junto a *sparePopulation* para uma eventual busca por novidade, condizendo com o modelo em que todos os indivíduos da população reserva e principal competem pelo título de solução mais inovadora. Visto pelo lado da inovação, indivíduos repetidos podem distorcer a medida de inovação, por isso, *sparePopulation* é filtrada, removendo soluções repetidas.

4.3 CONSIDERAÇÕES FINAIS

As diferenças entre a abordagem original e a implementação proposta residem no mecanismo de disparo utilizado e nas ações subsequentes. Na versão original, o disparo do mecanismo de novidade é baseado em probabilidades, onde 8 indivíduos da população B com maior novidade são selecionados e substituem indivíduos com pior fitness em A, enquanto as demais operações permanecem inalteradas. Já na implementação proposta, o disparo do mecanismo é determinado pela métrica de estagnação da população, medida pelo número de gerações em que o melhor fitness não melhorou. Além disso, foram introduzidas variações na

condição de disparo, como a redução do limite pela metade e a inclusão de uma população gerada aleatoriamente na busca por novidade, onde 8 indivíduos são selecionados da população reserva e substituem aleatoriamente indivíduos na população principal. Essas diferenças representam as principais distinções entre o método original e a implementação proposta.

As implementações do GA e GA_{NS} foram baseadas nos esquemas propostos na literatura definidos no capítulo 2, utilizando a inicialização da população com o método de busca gulosa, seleção de parentes por torneio, critério de parada um número limite de gerações e métrica de qualidade a menor distância de cada indivíduo. Os operadores genéticos para ambos foram realizados de forma que respeitem as restrições do TSP. Além disso, foi descrita e definida a implementação do mecanismo de disparo, avaliação e inserção de novidade.

No próximo capítulo, serão apresentados os resultados obtidos com a aplicação do GA e GA_{NS} ao TSP, além de uma análise dos resultados e comparação com outros métodos.

5 EXPERIMENTOS

5.1 MODELAGEM DO EXPERIMENTO

Nos capítulos anteriores, foram levantadas as definições do TSP e suas aplicação e métodos de resolver o problema. Dentre esses métodos, estão a utilização de algoritmos bio-inspirados, mais especificamente o Algoritmo Genético e Novelty Search. Foi proposta a utilização do Novelty Search em conjunto com o Algoritmo Genético com o intuito de melhorar seus resultados. Sendo assim, é necessário que experimentos sejam conduzidos para comprovar a eficácia, determinar quais cenários são benéficos e o custo-benefício de utilizar esse modelo proposto.

Mais especificamente, o experimento tem o objetivo de comparar o desempenho entre o GA tradicional com o GA_{NS} (Algoritmo Genético com Novelty Search) para a resolução do TSP. O benchmark utilizado é o TSPLib (19), onde as instâncias selecionadas são encontradas na tabela 5.1. Cada instância possui suas particularidades que podem impactar o algoritmo utilizado para solucionar, como por exemplo o número de cidades, formato das distâncias, quantidade de objetivos e variar o tipo de grafo, como ser completo ou não.

Sabendo que resultados podem variar de acordo com a inicialização das instâncias e a semente aleatória, duas alternativas podem ser escolhidas ou utilizadas em conjunto. A primeira seria utilizar a mesma semente aleatória e parâmetros para a execução tanto do GA quanto do GA_{NS}. A segunda seria realizar várias execuções para cada instância do TSPLib e comparar as médias dos dados obtidos. A terceira seria a combinação das duas alternativas anteriores, onde seriam realizadas várias execuções para ambos os algoritmos com os mesmos parâmetros, porém utilizando a mesma semente para cada execução, modificando-as a cada execução. Para esse experimento, a terceira opção foi escolhida.

Utilizando os mesmos parâmetros informados na lista a seguir, é possível reproduzir os resultados encontrados, uma vez que, o algoritmo terá o mesmo comportamento dada a semente aleatória utilizada pelo algoritmo. Os parâmetros utilizados para as execuções dos dois algoritmos foram:

- $PopulationSize$ = 50: Tamanho da população;
- $NumIterations$ = 30000: Número de iterações;
- $crossoverRate$ = 0.7: Taxa de crossover;
- $mutationRate$ = 0.2: Taxa de mutação; e
- $noveltyIters$ = 100: Número de iterações para disparo do NS.
- $seeds$: 1678007458 334458702 1682404983 4014355689 2653289983 1860403883 3247074786 1174563364 3186218264 2888613824 2786328941 3008427978 603485389 1328469719 2931301805 4215399172 4101105735 3109312980 1869554770 855791490 1668498819 2533100885 456833438 2814657188 832278848 1394861728 3445500042 1527472704 3247737027 1581348115.

Visando as boas práticas levantadas em (15) para a implementação dos algoritmos, no capítulo 4 foram apresentados todos os detalhes, restrições e parâmetros utilizados para a

execução dos métodos. Seguindo para as boas práticas em relação a apresentação dos dados, como mencionado, são utilizadas os mesmos parâmetros e instâncias para ambos os códigos utilizados. Seus respectivos tempos de execução, tempo de convergência, melhor e pior resultados, média e desvio padrão serão mostrados na seção Resultados.

5.1.1 TSPLib

TSPLib é uma biblioteca de instâncias de benchmark para o TSP. Ele foi desenvolvido para fornecer um conjunto padrão de instâncias de teste para avaliar e comparar o desempenho de diferentes algoritmos que resolvem o TSP. Com isso, é possível contornar o desafio de comparar o desempenho de diferentes algoritmos, pois não há uma maneira direta de determinar qual algoritmo é melhor.

TSPLib fornece um conjunto padronizado de instâncias de benchmark com soluções ótimas conhecidas que podem ser usadas para avaliar e comparar o desempenho de diferentes algoritmos. Dessas instâncias fornecidas, o número de cidades variam, tornando o problema mais complexo ou simples. Por padrão, o formato estabelecido para os nomes das instâncias segue *Inst**YYYY***, onde ***YYYY*** *representa o número de vértices do grafo e Inst o nome do conjunto de pontos do grafo. Sendo assim, a instância Berlin52 possui 52 cidades e os pontos representam uma viagem por Berlin.*

Usar TSPLib como benchmark tem várias vantagens. Primeiro, ele fornece um conjunto padronizado de instâncias de teste que podem ser usadas para comparar o desempenho de diferentes algoritmos de maneira consistente. Segundo, ele fornece soluções ótimas conhecidas, então o desempenho de um algoritmo pode ser avaliado com precisão comparando suas soluções às soluções ótimas. Terceiro, ele fornece uma grande variedade de instâncias TSP com diferentes características, como diferentes números de cidades, diferentes distâncias entre cidades e diferentes níveis de simetria, o que permite uma avaliação abrangente do desempenho de um algoritmo.

5.1.2 Ambiente de teste

Para garantir a consistência e a comparabilidade dos experimentos, todos eles foram conduzidos em um ambiente padronizado. Essa configuração virtualizada, utilizada como base, é composta por Quatro CPUs AMD EPYC 7401 com a frequência de operação é de 1999.999 MHz, cada uma com 24 núcleos e memória do sistema de 200GiB, Ao utilizar esse ambiente consistente e bem dimensionado, foi possível obter resultados confiáveis e significativos, contribuindo para a compreensão e aprimoramento da técnica em estudo.

5.1.3 Acesso ao código

O código implementado neste trabalho está disponível para acesso no repositório GitHub, no seguinte endereço: ga-ns-tsp-solver. Nesse repositório, é possível encontrar toda a estrutura do projeto, incluindo os arquivos de código-fonte, bibliotecas utilizadas e qualquer documentação adicional relacionada ao desenvolvimento do sistema. Além disso, para a obtenção das instâncias do problema do TSP (Traveling Salesman Problem), recomenda-se visitar o site TSPlib (www.tsplib.com), que é uma biblioteca amplamente reconhecida e utilizada para compartilhamento de instâncias do TSP. Nesse site, é possível encontrar uma vasta coleção de instâncias de diferentes tamanhos e características, permitindo realizar testes e avaliações de desempenho de algoritmos para o TSP.

5.2 RESULTADOS

Essa seção tem como objetivo apresentar os resultados obtidos experimentalmente através da metodologia proposta anteriormente. Para os dois algoritmos implementados, foi adotado 30.000 iterações como critério de parada. Os resultados serão mostrados através das tabelas de fitness e de tempo.

A tabela de fitness 5.1, apresenta a média dos resultados obtidos na coluna **Média**, melhores resultados encontrados na coluna **Melhor** e piores resultados na coluna **Pior**. A coluna **Melhor** apresenta dentre as 30 execuções da instância qual foi o melhor resultado obtido. Analogamente, a coluna de pior resultado apresenta o pior resultado obtido. Já a coluna ótimo, é apresentado o melhor valor que é possível obter para a instância, sendo que não existe valor menor ao apresentado na coluna. As células da tabela de um algoritmo destacados em negrito significam que tem resultado melhor em relação ao outro algoritmo.

Já a tabela de tempos de convergência 5.2, contém as informações relacionadas ao tempo de convergência medidos em segundos ($T_{Conv(s)}$) e medidos em iterações(It_{Conv}) juntamente com o desvio padrão (Desvio) das iterações. O tempo de convergência é dado pela iteração na qual o algoritmo encontrou o melhor resultado de sua execução. Sendo assim, se o melhor resultado de um algoritmo para uma instância foi encontrado na geração 10, o tempo de convergência será a diferença entre o tempo da geração 10 e o início da primeira geração. Lembrando que são executados para cada algoritmo 30 execuções para cada instância, a tabela de convergência mostra a média do tempo de convergência para cada iteração. Uma média do tempo de convergência baixa significa que ou o algoritmo encontrou a melhor solução possível em poucas iterações, ou que o algoritmo estagnou em um ótimo local muito antecipadamente e a exploração não obteve sucesso em encontrar melhores soluções.

São também apresentadas as tabelas de tempo de execução 5.3 e de comparação com a literatura. A tabela de tempo de execução apresenta a comparação dos tempos de execução dos dois algoritmos através das colunas de tempo de execução em segundos (Tempo(s)), desvio padrão do tempo de execução (Desvio) e número de ativações do mecanismo de Fitness(Ativações). Por fim, a tabela de comparação com a literatura traz a comparação do Novelty Search desenvolvido com as aplicações feitas na literatura, com os dados apresentados sendo retirados dos trabalhos publicados.

5.2.1 Fitness e convergência

Os resultados obtidos das execuções de ambos algoritmos podem ser vistos na tabela 5.1. Para todas as instâncias é possível verificar que o algoritmo GA_{NS} obteve na média melhores soluções em relação ao GA. A tabela apresenta também o melhor e o pior resultado apresentado nas 30 execuções de cada algoritmo. Para tais dados, a maioria das instâncias o GA_{NS} apresentou melhores resultados em relação ao GA para melhor e pior fitness devolvidos. Em média, a aplicação do Novelty Search ao GA, para as instâncias executadas, melhorou a média dos resultados em 2,16%. A melhoria mais significativa obtida foi para a instância Eil101, com 5,58% e a menos relevante para o KroA100 com melhora de 0,15%. O GA_{NS} também apresentou uma convergência mais tardia. Isso, aliado à melhora na média dos resultados, aponta que o algoritmo se beneficia mais de possuir um maior número de gerações, mostrando que possui uma capacidade maior de exploração do espaço, o que era esperado.

Para a instância Eil101, temos a melhora do fitness para o melhor e o pior resultado encontrado, isso acontece devido a maior exploração proporcionada pela novidade como pode ser vista na tabela 5.2, onde, na média, a convergência do algoritmo é muito mais tardia que no GA, tomando maior proveito de uma maior quantidade de gerações. Porém para o caso da instância

KroA100, o ganho com a novidade é praticamente imperceptível, possuindo um desvio padrão menor mas com uma distância interquartil praticamente idêntica. Para instâncias menores, como o Burma14, o GA_{NS} apresentou uma taxa maior de acertos para melhor fitness e para instâncias médias, como St70, apresentou um ganho baixo. Com isso, não é possível afirmar que o tamanho da instância influencia no melhor resultado, podendo apenas afirmar que o ganho é dependente da instância executada.

Tabela 5.1: Valores coletados dos algoritmos GA e GA_{NS} a partir das instâncias do TSPlib.

		GA_{NS}			GA		
Instâncias	Ótimo	Média	Melhor	Pior	Média	Melhor	Pior
burma14	3323	**3387,93**	**3323**	**3756**	3453,47	**3323**	4054
eil51	426	**538,80**	500	**590**	556,63	**489**	656
berlin52	7542	**9829,90**	**8686**	**10743**	10047,40	9158	11317
st70	675	**47498,83**	**45505**	**50548**	48423,33	46248	51742
eil76	538	**730,00**	**643**	**899**	742,70	660	883
kroA100	21282	**39349,07**	**32498**	**47308**	39409,37	33969	47482
eil101	629	**856,00**	**741**	**954**	906,63	826	1023
pr107	44303	**47498,83**	**45505**	**50548**	48423,33	46248	51742
pr124	59030	**74826,53**	67265	88389	76294,93	**63855**	**86775**

Tabela 5.2: Tempo de convergência em iterações (It_{Conv}) e em segundos (T_{Conv})

	GA_{NS}			GA		
Instâncias	It_{Conv}	Desvio	$T_{Conv}(s)$	It_{Conv}	Desvio	$T_{Conv}(s)$
burma14	6334	2065	140,93	7408	10162	76,25
eil51	20494	8792	971,37	10910	10347	326,48
berlin52	22543	8150	1130,55	6562	9030	206,36
st70	17789	9777	1036,92	12899	8734	519,01
eil76	21211	7840	1023,64	12478	8867	550,59
kroA100	19850	8060	1109,85	13618	9997	731,63
eil101	20098	7940	1175,71	15210	7828	839,16
pr107	15424	9775	1015,88	8633	8829	511,66
pr124	13694	8663	1027,72	8064	4579	652,79

5.2.2 Boxsplot

Em complemento à seção anterior, serão apresentados os boxplots das execuções realizadas, boxsplots são uma representação gráfica que exibe a distribuição de um conjunto de dados estatísticos, destacando medidas-chave, como a mediana, quartis e a presença de possíveis valores atípicos (outliers), que ajudam a entender informações sobre o comportamento e a estabilidade das execuções. Na figura 5.1 é apresentado um gráfico para cada instância e em cada gráfico o boxplot de cada algoritmo.

Ao analisar os gráficos de comparação entre o GA (Algoritmo Genético) e o GA_{NS} (Algoritmo Genético com Busca por Novidade), em vários casos a distância interquartil é menor no GA_{NS}, como evidenciado pelos gráficos de boxplot, com exceção das instâncias Eil51 e Eil76. Sabendo que para a menor e menor instância testada a diferença entre o quartil inferior e

superior é na maioria dos casos menor, é possível afirmar que o NS traz maior estabilidade aos resultados independente do tamanho da instância. Foi observado também uma queda na mediana dos resultados, indicando uma tendência a possuir resultados melhores. Essas diferenças se devem à aplicação da busca por novidade, que permite à população manter um nível adequado de diversidade e, posteriormente, encontrar resultados mais próximos. A utilização da busca no espaço comportamental permite que o algoritmos resista a manter os indivíduos de uma população se tornem excessivamente semelhantes entre si, promovendo uma busca mais ampla, que acarreta em soluções mais próximas, trazendo o equilíbrio nos quartis.

Esse comportamento acontece devido a maior capacidade de exploração do espaço de busca, já que com uma busca mais ampla, é mais provável de encontrar soluções melhores. Vale ressaltar que é uma busca direcionada e que a seleção dos indivíduos continua sendo a mesma realizada pelo GA tradicional implementado.

5.2.3 Tempo de execução

Seguindo a recomendação da literatura, nessa subseção são discutidos os resultados obtidos pelo tempo de execução dos algoritmos. É apresentado também a número de ativações do mecanismo de novidade realizadas. Esses dados são encontrados na tabela 5.3. O tempo de execução é dado pela diferença entre o tempo em que a última geração é calculada e o tempo em que é iniciada a primeira geração. A apresentação do tempo de execução dos algoritmos nem sempre é considerada na literatura. Porém, por se tratar de um problema de otimização é importante que estes dados sejam apresentados para mostrar os custos de cada aplicação. Foi decidido mostrar juntamente nessa tabela a quantidade de ativações do mecanismo de novidade, uma vez que, é o custo adicional da aplicação da novidade no GA.

Como o esperado, o GA foi mais rápido em praticamente todas as execuções, salvo a instância Pr124. Porém, dado os desvios padrões altos para essa instância, podemos assumir que ambos algoritmos tiveram praticamente o mesmo desempenho. O fato do GA possuir maior velocidade de execução que o GA_{NS} é algo esperado devido ao fato de que em cada disparo do mecanismo de novidade, o GA_{NS} faz no mínimo o dobro mais um de avaliações de população. Em média, o GA_{NS} avalia juntamente com a população principal e reserva, 4220 populações aleatórias durante uma execução de 30000 gerações. Com isso, é possível avaliar o custo de se implementar o algoritmo, onde, por mais que seja otimizado o código, o mecanismo de novidade irá adicionar complexidade extra quando utilizado em conjunto com heurísticas tradicionais.

Tabela 5.3: Tempo de execução em segundos dos algoritmos GA e GA_{NS} e número de ativações do mecanismo de novidade.

	GA_{NS}			GA	
Instâncias	Tempo (s)	Desvio	Ativações	Tempo (s)	Desvio
burma14	697,46	15,99	14389	310,73	6,45
eil51	1446,97	54,75	13037	898,60	7,17
berlin52	1515,58	29,61	13190	942,73	7,17
st70	1802,17	45,00	12600	1207,73	10,97
eil76	21210,97	7839,56	12044	7407,93	10161,84
kroA100	1687,75	36,74	11165	1605,11	28,76
eil101	1770,45	26,72	11050	1655,44	26,49
pr107	1983,24	44,84	12535	1780,02	36,61
pr124	2280,55	26,90	11701	2385,75	177,22

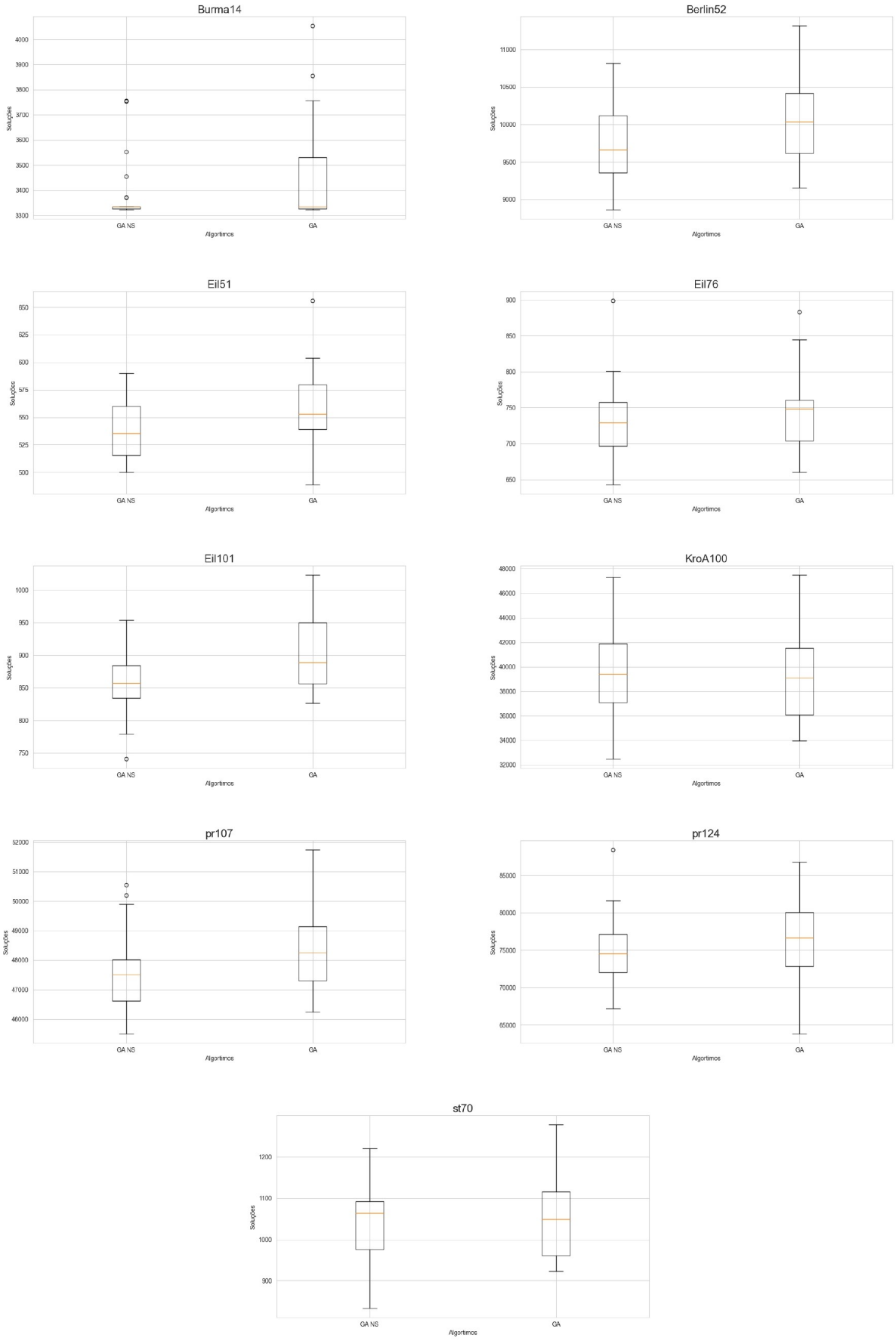


Figura 5.1: Boxplot dos resultados dos algoritmo GA e GA_{NS} para as instâncias testadas.

5.2.4 Resultados da literatura

Com os dados publicados no artigo (16) é possível relacionar os ganhos obtidos do GA_{NS} com a aplicação original. Embora o Novelty Search não tenha sido aplicado ao GA especificamente, é possível realizar uma medida relativa através da comparação entre um algoritmo original e o algoritmo aplicado. A aplicação do Novelty Search realizada nesse trabalho de graduação é condizente com os resultados obtidos por Osaba em seu paper.

Em suma, a melhora média dos resultados obtida nesse experimento foi de 2,16%. Osaba realizou a aplicação em três diferentes algoritmos, obtendo melhora média de 1,95% para o Particle Swarm Algorithm, 0,65% para o Firefly Algorithm e 2,24% para o Bat Algorithm. Porém, o autor relata um aumento na velocidade de convergência com a aplicação do Novelty Search, o que foi observado o contrário nesse experimento. Os valores separados por instância podem ser vistos na tabela 5.4. Valores negativos significam uma perda de desempenho em relação ao algoritmo original.

Vale ressaltar que, embora o algoritmo implementado nesse trabalho seja baseado nas descrições do autor, não são o mesmo algoritmo e possuem diferenças citadas no capítulo 4.

Tabela 5.4: Comparação de ganho da aplicação do Novelty Search entre esse trabalho e a literatura. Valores coletados em (16).

	GA_{NS}	PSO_{NS}	FA_{NS}	BA_{NS}
eil51	3,20%	0,59%	0,56%	0,80%
berlin52	2,16%	-0,42%	1,11%	1,18%
st70	0,96%	0,55%	0,77%	1,19%
eil76	1,71%	1,38%	0,65%	1,34%
kroA100	0,15%	1,91%	0,69%	3,50%
eil101	5,58%	5,76%	1,18%	2,37%
pr107	1,91%	1,82%	0,85%	2,18%
pr124	1,92%	5,87%	3,07%	6,03%

6 CONCLUSÃO

Este trabalho consiste da exploração de conceitos de meta-heurísticas bioinspiradas, suas modelagens e aplicações ao TSP. Mais especificamente, foram estudas duas abordagens: o Algoritmo Genético e o Novelty Search. A ideia é utilizar da busca no espaço comportamental provida pelo Novelty Search para aumentar a capacidade de exploração e resistência a ótimos locais do Algoritmo Genético. A partir dos conceitos apresentados na literatura, ambos algoritmos foram modelados, implementados e modificados com o objetivo de obter uma melhora dos resultados. Os testes para ambos algoritmos foram realizados utilizando os mesmos parâmetros e número de geração, com o TSPlib sendo utilizado como benchmark.

Na análise dos resultados e obtenção de informações, foram empregados gráficos e plotagens ao longo das iterações, bem como boxplots. Além disso, foi elaborada uma tabela contendo as informações de cada execução, como média, desvio padrão e melhores e piores resultados, além da comparação dos tempos de convergência. Essas medidas permitiram uma avaliação comparativa entre o GA (Algoritmo Genético) e o GA_{NS} (Algoritmo Genético com Busca por Novidade), proporcionando uma visão abrangente do desempenho e das características distintas de cada abordagem. Foi possível obter uma melhora de 5,58% do GA para o melhor caso e uma melhora de 2,16% na média.

Outro ganho obtido com o Novelty Search foi uma maior estabilidade e um maior aproveitamento de um número maior de gerações por execução. Os resultados obtidos foram também comparados com a literatura, embora o NS não tenha sido aplicado no GA, é possível comparar o ganho relativo, e foi concluído que os resultados desse trabalho estão condizentes com a literatura. Com estas informações obtidas experimentalmente, pode-se concluir que os benefícios esperados segundo a pesquisa foram atingidos com a utilização da busca no espaço comportamental para o Algoritmo Genético.

Por se tratar de uma aplicação pouco documentada na literatura, foi necessário realizar toda a modelagem e implementação dos algoritmos. Isso levou a algumas restrições para o trabalho. Uma dessas restrições é a exploração em relação aos impactos que cada parâmetro possui no desempenho dos algoritmos. Modificar um parâmetro pode comprometer completamente ou melhorar muito o desempenho, e foi decidido executar os algoritmos com os mesmos parâmetros, escolhidos como ótimo empiricamente.

Outra limitação é o tamanho das instâncias, quanto maior o número de cidades, maior é o custo computacional para o experimento. Porém, utilizar instâncias com um número maior de cidades, por exemplo 1000 cidades, pode trazer informações importantes em relação ao comportamento dos algoritmos, como se a capacidade exploratória do Novelty Search pode ajudar ou atrapalhar a convergência do algoritmo.

Como expansão para o trabalho, é possível realizar a aplicação do modelo do Novelty Search proposto em outros algoritmos baseados em populações, como o Particle Swarm Optmization e o Ant Colony Optmization. Pode-se também, aplicar o modelo a um Algoritmo Genético otimizado para solucionar o TSP, dessa forma, tornaria possível ver os ganhos e perdas da utilização do Novelty Search quando o algoritmo original em que é aplicado se aproxima mais vezes do resultado ótimo. Outra expansão é a utilização de testes estatísticos mais robustos, como testes não-paramétricos, para a validação dos dados levantados nos testes realizados com o objetivo de confirmar estatisticamente as melhoras nos resultados.

REFERÊNCIAS

[1] Beasley, D., Bull, D. R. e Martin, R. R. (1993). An overview of genetic algorithms: Part 1, fundamentals. *University computing*, 15(2):56–69.

[2] Deng, Y., Liu, Y. e Zhou, D. (2015). An improved genetic algorithm with initial population strategy for symmetric tsp. *Mathematical Problems in Engineering*, 2015.

[3] Goldberg, D. E. e Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. Em *Foundations of genetic algorithms*, volume 1, páginas 69–93. Elsevier.

[4] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C. e Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

[5] Helsgaun, K. (2000). An effective implementation of the lin–kernighan traveling salesman heuristic. *European journal of operational research*, 126(1):106–130.

[6] Holland, J. H. (1992). Genetic algorithms computer programs that"evolve"in ways that resemble natural selection can solve complex problems even their creators do not fully understand. *Scientific American*, páginas 66–72.

[7] Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50.

[8] Lehman, J. e Stanley, K. O. (2011a). Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223.

[9] Lehman, J. e Stanley, K. O. (2011b). Novelty search and the problem with objectives. *Genetic programming theory and practice IX*, páginas 37–56.

[10] Lehman, J. e Stanley, K. O. (2013). Evolvability is inevitable: Increasing evolvability without the pressure to adapt. *PloS one*, 8(4):e62186.

[11] Lehman, J., Stanley, K. O. et al. (2008). Exploiting open-endedness to solve problems through the search for novelty. Em *ALIFE*, páginas 329–336.

[12] Li, X., Xiao, N., Claramunt, C. e Lin, H. (2011). Initialization strategies to enhancing the performance of genetic algorithms for the p-median problem. *Computers & Industrial Engineering*, 61(4):1024–1034.

[13] Mzili, T., Mzili, I. e Riffi, M. E. (2023). Artificial rat optimization with decision-making: A bio-inspired metaheuristic algorithm for solving the traveling salesman problem. *Decision Making: Applications in Management and Engineering*.

[Osaba et al.] Osaba, E., Carballedo, R., Diaz, F., Onieva, E., de la Iglesia, I. e Perallos, A. Crossover vs. mutation: A comparative analysis of the evolutionary strategy of genetic algorithms applied to combinatorial optimization problems.

[15] Osaba, E., Carballedo, R., Diaz, F., Onieva, E. e Perallos, A. (2014). A proposal of good practice in the formulation and comparison of meta-heuristics for solving routing problems. Em *International Joint Conference SOCO'14-CISIS'14-ICEUTE'14: Bilbao, Spain, June 25th-27th, 2014, Proceedings*, páginas 31–40. Springer.

[16] Osaba, E., Yang, X.-S. e Del Ser, J. (2020). Traveling salesman problem: a perspective review of recent research and new results with bio-inspired metaheuristics. *Nature-Inspired Computation and Swarm Intelligence*, páginas 135–164.

[17] Rao, A., Hegde, S. K., Rao, A., Hegde, K., Rao, I., Hegde, K., Rao, A. e Hegde, S. (2015). Literature survey on travelling salesman problem using genetic algorithms. *International Journal of Advanced Research in Eduation Technology (IJARET)*, 2(1):42.

[18] Razali, N. M., Geraghty, J. et al. (2011). Genetic algorithm performance with different selection strategies in solving tsp. Em *Proceedings of the world congress on engineering*, volume 2, páginas 1–6. International Association of Engineers Hong Kong, China.

[19] Reinelt, G. (1991). Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384.

[20] Scholz, J. (2019). Genetic algorithms and the traveling salesman problem a historical review. *arXiv preprint arXiv:1901.05737*.

[21] Starkweather, T., McDaniel, S., Mathias, K. E., Whitley, L. D. e Whitley, C. (1991). A comparison of genetic sequencing operators. Em *ICGA*, páginas 69–76.

[22] Sun, L. (2015). Genetic algorithm for tsp problem. Em *2015 International Industrial Informatics and Computer Engineering Conference*, páginas 1436–1439. Atlantis Press.

[23] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P. e SciPy 1.0 Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272.

[24] Wei, Y., Hu, Y. e Gu, K. (2007). Parallel search strategies for tsps using a greedy genetic algorithm. Em *Third international conference on natural computation (ICNC 2007)*, volume 3, páginas 786–790. IEEE.