Universidade Federal Rural de Pernambuco

Departamento de Estatística e Informática

Bacharelado em Sistemas de Informação

FlyFood

José Roberto Oliveira de Araújo Filho

Recife

Março de 2023

Resumo

Dado o grande fluxo de automóveis nas cidades e o alto custo com mão-de-obra, há uma oportunidade para que empresas de delivery utilizem drones como veículos de entrega, de forma que realizem suas entregas mais rapidamente e tenham um custo menor. A partir disso, a FlyFood surgiu com o objetivo de desenvolver um algoritmo que calcule as distâncias dos pontos de entrega e encontre a rota de menor custo, a fim de otimizar os trajetos dos drones. Para tal, foi implementada uma metaheurística de busca e otimização capaz de encontrar uma boa solução em tempo polinomial, chamada de algoritmo genético, utilizando substituição geracional por elitismo. Contudo, por se tratar de uma metaheurística, ao utilizarmos o AG nós abrimos mão da garantia de encontrar a solução ótima.

Palavras-chave: Algoritmo; Metaheurística; Otimização; Algoritmo genético.

1. Introdução

1.1. Apresentação e motivação

Ao longo dos anos, as pessoas têm preferido cada vez mais fazer suas compras de forma online, evitando perder tempo indo até a loja física e esperando na fila para realizar suas compras, além de economizar o custo com o transporte. Com isso, as empresas de e-commerce vêm investindo gradativamente em caminhões, carros e vans para realizarem suas entregas mais rapidamente. Contudo, devido ao aumento do fluxo de automóveis nas cidades, as empresas de delivery vem enfrentando dificuldades para realizarem suas entregas, tendo um custo mais alto com entregadores por causa da mão-de-obra e não conseguindo fazer suas entregas em um tempo adequado.

Visto que entregas por vias terrestres têm sido um problema, observa-se uma oportunidade de solucionar tal questão utilizando drones para realizarem as entregas por via aérea. Apesar do drone não ter a capacidade de transporte de um automóvel - como um caminhão ou uma van, por exemplo -, ele mostra-se como uma maneira da empresa prestar seus serviços de forma mais eficiente - diminuindo o custo com entregadores, combustível, e o tempo gasto no trânsito.

Dada a alta demanda da sociedade por serviços de delivery e a dificuldade cada vez maior de locomoção por via terrestre, é necessária a implementação de uma alternativa por meio da tecnologia, a fim de: contribuir para a modernização dos serviços prestados pelas empresas; para desobstrução do tráfego nas cidades; e para que os clientes recebam suas mercadorias mais rapidamente; permitindo assim o aumento de produtividade de toda a sociedade e a evolução da tecnologia em busca de algoritmos e métodos mais eficientes.

1.2. Formulação do problema

Este problema tem como entrada uma matriz, na qual **P** = $\{p_1, p_2, ..., p_n\}$ representa o conjunto com ***n*** pontos ***p***. As coordenadas de cada ponto são dadas pela sua posição **(*x, y*)** no plano cartesiano;

e a distância entre cada ponto é dada por $\left(d_{ab}\right)$, onde $d_{ab} = d\left(p_a, p_b\right)$, que é a distância de p_a para p_b.

O objetivo aqui é otimizar a rota dos drones, ou seja, queremos minimizar o somatório das distâncias d_{ab}, de forma que consigamos encontrar o menor caminho. Logo, o nosso problema pode ser representado matematicamente por:

$$min \sum d_{ab}$$

1.3. Objetivo

Desenvolver um algoritmo que calcule as distâncias dos pontos de entrega em um plano cartesiano e encontre a rota de menor custo, de forma que otimize os trajetos dos drones para que eles realizem entregas mais rápidas e o máximo de entregas possíveis dentro da sua capacidade de bateria.

1.4. Organização do trabalho

Daqui em diante, o trabalho está organizado da seguinte forma: A Seção 2 apresenta o embasamento teórico para entender o problema e a solução desenvolvida. Três trabalhos relacionados a este são apresentados na Seção 3, com uma breve descrição de como eles funcionam e suas limitações. A metodologia utilizada para solucionar o problema deste trabalho está presente na Seção 4. As Seções 5 e 6 apresentam os experimentos realizados com o algoritmo desenvolvido e os resultados obtidos neste trabalho, respectivamente. As conclusões do trabalho estão na Seção 7.

2. Referencial teórico

Esta seção apresenta a base teórica necessária para compreender o tipo de problema que estamos lidando, como é feita a verificação de eficiência de um algoritmo e procedimentos de busca e otimização. Sendo assim, aqui contém explicações da análise de algoritmos, das classes de problemas computacionais, de metaheurísticas e algoritmos genéticos - sendo eles pilares fundamentais para o entendimento da solução desenvolvida e das limitações do problema.

2.1. Análise de algoritmos

A análise de algoritmos é utilizada para pressupor os recursos necessários para que um determinado algoritmo seja executado - dentre esses recursos, os que mais buscamos mensurar são a memória e o tempo. O algoritmo mais eficiente é encontrado após fazer esta análise com vários outros algoritmos que resolvem o mesmo problema, e, por vezes, é possível até encontrar mais de um algoritmo eficiente (CORMEN, 2012).

De acordo com Cormen (2012), tendo em vista que o tempo que o algoritmo gasta para resolver um problema cresce de acordo com o tamanho da entrada, e, além disso, depende da máquina em que ele está sendo executado e da linguagem de programação utilizada, a eficiência de um algoritmo se dá pela quantidade de passos que ele executa para processar uma entrada de tamanho *n* - em que a entrada e *n* dependem do problema. Portanto, com a análise de algoritmos nós podemos saber quanto tempo um algoritmo leva para processar uma entrada de tamanho *n* (FEOFILOFF, 2015).

Quando vamos medir a eficiência de um algoritmo pelo tempo que ele despende para resolver um problema com uma entrada de tamanho *n*, nós utilizamos a análise assintótica. Essa análise nos diz qual a ordem de crescimento do tempo de execução de um algoritmo quando a entrada é muito grande (BRUNET, 2019).

A principal notação assintótica usada é a Notação O (Big O notation), pois ela define o limite superior do crescimento do tempo de execução de um algoritmo no pior caso. Ou seja, a Big O notation nos fornece o tempo de execução máximo de um algoritmo quando o valor de *n* da entrada é muito grande.

Por exemplo: Merge Sort é um algoritmo que ordena os elementos de uma estrutura de dados, geralmente chamada de array. A notação O desse algoritmo é O($n \log_2 n$), ou seja, o tempo de execução máximo que o Merge Sort leva para ordenar, no pior caso, um array com *n* elementos, é igual a $n \log_2 n$.

Sendo assim, o algoritmo mais eficiente é aquele que tem o menor Big O notation em comparação com outros algoritmos que buscam resolver o mesmo problema. Contudo, vale ressaltar que:

> Devido a fatores constantes e termos de ordem mais baixa, um algoritmo cujo tempo de execução tenha uma ordem de crescimento mais alta pode demorar menos tempo para pequenas entradas do que um algoritmo cuja ordem de crescimento seja mais baixa. Porém, para entradas suficientemente grandes, um algoritmo $\theta(n^2)$, por exemplo, será executado mais rapidamente no pior caso que um algoritmo $\theta(n^3)$. (CORMEN, 2012).

2.2. Classes de problemas computacionais

Os problemas computacionais são divididos em classes de acordo com a sua complexidade, ou seja, de acordo com o tempo de execução do algoritmo no pior caso. Além disso, os problemas são divididos em *tratáveis* e *intratáveis*. Os problemas tratáveis, ou fáceis, são aqueles que um algoritmo resolve em tempo polinomial; já os problemas intratáveis, ou difíceis, são problemas que o algoritmo não consegue resolver em tempo polinomial. As classes de problemas que serão descritas a seguir serão a P, NP, NP-Completo e NP-Difícil, mas vale ressaltar que elas não são as únicas existentes.

A **classe P** compreende os problemas considerados como fáceis ou tratáveis, ou seja, problemas que podem ser resolvidos em tempo polinomial. Exemplos: problema do caminho mínimo e problema de ordenação.

A **classe NP** consiste nos problemas que são verificáveis em tempo polinomial, isto é, dada uma solução, conseguimos verificar em tempo polinomial se esta solução é válida ou não (OLIVETE, 2020?). Exemplo: problema do caminho Hamiltoniano.

O problema pertence à **classe NP-Completo** se ele também pertence à classe NP, destacando que, se existir um algoritmo que resolva um problema NP-Completo em tempo polinomial, todos os problemas NP também serão resolvidos em tempo polinomial. Exemplo: problema do circuito Hamiltoniano em grafo orientado.

Um problema *A* pertence à **classe NP-Difícil** se existir um problema NP-Completo *B*, de forma que *B* seja redutível à *A* em tempo polinomial (ACERVO LIMA, c2022). Assim sendo, um problema é NP-Difícil se ele for tão difícil quanto um problema NP-Completo. Exemplo: problema do caixeiro viajante. A diferença entre um problema NP-Completo e NP-Difícil é que, o problema é NP-Completo se ele fizer parte da classe NP e NP-Difícil; e para o problema ser

NP-Difícil ele tem que ser tão difícil quanto um problema NP, mas não necessariamente pertencer à classe NP (ACERVO LIMA, c2022).

2.3. Problema do Caixeiro Viajante

O problema do caixeiro viajante (PCV) consiste em encontrar a rota de menor distância em um grafo, partindo de uma cidade (vértice) de origem, passando por várias outras cidades uma única vez e retornando à cidade inicial. O custo da rota é calculado somando a distância (peso) entre cada vértice adjacente de um determinado caminho.

O problema deste trabalho é NP-Difícil, visto que ele é semelhante ao PCV. No entanto, é necessário elucidar que, até o momento em que este trabalho foi realizado, não foi descoberto nenhum algoritmo capaz de resolver problemas NP-Completo e NP-Difícil em tempo polinomial - mas também não foi provado que nenhum algoritmo desse tipo exista.

2.4. Metaheurísticas

Metaheurística é um procedimento projetado para encontrar uma boa solução para problemas de otimização, especialmente com informações incompletas ou imperfeitas ou poder computacional limitado. Entretanto, apesar deste método encontrar boas soluções, não é garantido que ele encontrará a solução ótima do problema. Geralmente as metaheurísticas são usadas em problemas que não se conhece um algoritmo eficiente capaz de solucioná-lo, como os problemas NP-Completos e NP-Difíceis [6].

As metaheurísticas mais usadas são: GRASP; Subida de Encosta (Hill Climbing); Busca Tabu (Tabu Search); Algoritmo Genético (Genetic Algorithm); Enxame de Partículas (Particle Swarm); Colônia de Formigas (Ant Colony).

2.5. Algoritmos Genéticos (GA)

Algoritmo genético é um procedimento de busca e otimização desenvolvido por John Holland em 1975. Este método foi inspirado no processo de seleção natural que ocorre na natureza, conforme proposto por Charles Darwin. O AG é um algoritmo estocástico que trabalha com uma população de soluções simultaneamente, utilizando informações de custo e recompensa [7].

O algoritmo funciona da seguinte forma: inicialmente é gerada uma população inicial com *n* indivíduos, que representam possíveis soluções do problema. A partir disso, selecionamos, por roleta ou torneio, os pais de maior fitness (aptidão) para que seja efetuado o crossover (cruzamento) entre eles, de forma a produzir filhos que podem sofrer ou não mutação. Então, avaliamos a aptidão dos filhos gerados, que formam uma nova população, substituindo a população anterior. Esse processo é repetido até que o critério de parada seja atingido, podendo ele ser o número de gerações, convergência, entre outros [7, 8].

Sendo assim, o algoritmo genético parte de uma população inicial e vai evoluindo a população a cada nova geração, até que nos dê uma boa solução para o problema. Contudo, pelo fato do GA ser uma metaheurística, não há garantia de que essa solução seja a melhor.

3. Trabalhos relacionados

Aqui são apresentados três trabalhos relacionados a este, com uma breve descrição do que foi proposto e os resultados obtidos.

3.1. Solving TSP problem by using genetic algorithm

Este artigo propõe um novo método de representação de cromossomos usando matriz binária e um novo critério de aptidão para serem usados como método para encontrar a solução ótima para o PCV [9]. Neste trabalho foi fornecido um algoritmo para o PCV simétrico e assimétrico, uma nova representação de matriz assimétrica e foi aplicado crossover e mutação repetidas vezes a fim de obter a solução ideal.

Os resultados obtidos mostram que o procedimento desenvolvido para resolver o problema do caixeiro viajante simétrico e assimétrico usando algoritmos genéticos foi mais eficaz do que outros algoritmos do estado-da-arte.

3.2. Um algoritmo genético com infecção viral para o problema do caixeiro viajante

Este trabalho traz o algoritmo genético usando infecção viral para obter uma solução próxima da ótima em tempo polinomial para o problema do caixeiro viajante.

O método proposto no trabalho consiste em manter uma população extra, chamada 'População de Vírus', que contém trechos de soluções viáveis do problema. Nesse algoritmo é usado um operador de infecção, ao invés de um operador de mutação [10].

Os resultados mostram que o método proposto foi mais eficiente do que o algoritmo genético convencional, sendo melhor na qualidade das soluções e no tempo gasto para encontrá-las.

3.3. Meta-heurística Híbrida de Sistema de Colônia de Formigas e Algoritmo Genético para o Problema do Caixeiro Viajante

Este trabalho traz uma metaheurística híbrida de Sistema de Colônia de Formigas (ACS) e Algoritmos Genéticos para resolver o PCV, visando obter bons resultados.

No método proposto, quando o ACS não consegue sair de um ótimo local, ele exporta uma quantidade pré-determinada de soluções encontradas até o momento, a fim de formar a população inicial do algoritmo genético. A partir disso, o algoritmo de formigas é interrompido e o AG inicia para encontrar uma solução melhor do que o ACS. Depois que o algoritmo genético encontra a solução, ele exporta-a para o algoritmo de formigas adicionar feromônio no novo caminho, reforçando as alternativas de buscas para as formigas artificiais. Esse processo é repetido até que o critério de parada seja alcançado [11].

Os resultados obtidos mostram que o algoritmo híbrido proposto foi superior em todos os experimentos realizados, encontrando a solução ótima em todos eles e média das soluções próxima do valor ótimo.

4. Metodologia

Esta seção apresenta os pseudo-códigos e a explicação do algoritmo genético implementado para encontrar uma boa solução no problema do caixeiro viajante.

4.1. Introdução

A metodologia implementada (AG) consiste em encontrar uma boa solução para o PCV, partindo de uma população inicial com *n* indivíduos e realizando crossover e mutações com os indivíduos durante *m* gerações, sendo a quantidade de gerações o critério de parada do algoritmo. Foram seguidos os seguintes passos: (1) Gerar população inicial; (2) Calcular aptidão dos indivíduos; (3) Selecionar os pais; (4) Fazer crossover para gerar os filhos; (5) Fazer mutação dos filhos; (6) Calcular aptidão dos filhos gerados; (7) Selecionar os sobreviventes para a substituição geracional.

4.2. Passo 1: Geração da população inicial

Neste primeiro passo foi criada uma função responsável por gerar *n* caminhos a partir de uma lista contendo os pontos de entrega do arquivo de entrada. A quantidade de caminhos (indivíduos) é o tamanho da população.

Função 1: Gerar a população inicial

```
função
    função_1 (pontos: Lista[string], tamanho_pop: inteiro) :
Lista[lista]
var
    populacao : Lista
    inidividuo : Lista[string]
inicio
 1: populacao ← lista vazia
 2: para i de 0 até tamanho_pop faça
 3:     embaralhar lista de pontos
 4:     individuo ← lista embaralhada
 5:     se individuo não está em populacao então
 6:         populacao.append(individuo)
 7:     fimse
 8: fimpara
 9: retorne populacao
fimfunção
```

4.3. Passo 2: Cálculo da aptidão dos indivíduos

No segundo passo, a função criada é responsável por calcular a aptidão (custo do caminho) de todos os caminhos gerados na **função 1**. Para isso, foi criada uma função para calcular a aptidão de cada indivíduo.

Função 2: Calcular aptidão de um indivíduo

```
função
        função_2 (ind: Lista[string], coords: Dicionário[string, tupla]) :
float
var
        custo_caminho : float
        ponto_a : chave do dicionário
        ponto_b : chave do dicionário
inicio
 1: custo_caminho ← 0
 2: para i de 0 até len(ind) - 1 faça
 3:     ponto_a ← coords[ind[i]]
 4:     ponto_b ← coords[ind[i + 1]]
 5:     custo_caminho ← custo_caminho + sqrt((ponto_a[0] -
ponto_b[0])² + (ponto_a[1] - ponto_b[1])²)
 6: fimpara
 7: retorne custo_caminho
fimfunção
```

Função 3: Calcular aptidão da população

```
função
        função_3 (pop: Lista[lista], coordenadas: Dicionário[string,
tupla]) : Lista[float]
var
        aptidoes : Lista[float]
inicio
 1: aptidoes ← lista vazia
 2: para cada indivíduo em pop faça
 3:     aptidao ← função_2(individuo, coordenadas)
 4:     aptidoes.append(aptidao)
 5: fimpara
 6: retorne aptidoes
fimfunção
```

4.4. Passo 3: Seleção dos pais

Nesse passo foi criada uma função para fazer a seleção dos pais. Para tal, foi criada uma função que faz a seleção dos pais por torneio. A ideia aqui é criar uma lista de pais ordenada pela aptidão, começando do mais apto (menor caminho) e terminando no menos apto (caminho mais custoso). No entanto, não há garantia de que a lista dos pais esteja ordenada pela aptidão.

Função 4: Seleção por torneio

```
função
        função_4 (apt: Lista[float]) : int
var
        pai1 : int
        pai2 : int
inicio
 1: pai1 ← randint(0, (len(apt) - 1))
 2: pai2 ← randint(0 e (len(apt) - 1))
 3: se apt[pai1] < apt[pai2] entao
 4:     retorne pai1
 5: senão
 6:     retorne pai2
fimfunção
```

Função 5: Seleção dos pais

```
função
        função_5 (populacao: Lista[lista], aptidoes: Lista[float]) :
Lista[lista]
var
        lista_pais : Lista[lista]
        index : int
inicio
 1: lista_pais ← lista de len(populacao) listas preenchidas com None
 2: para i de 0 até len(populacao) faça
 3:     index ← função_4(aptidoes)
 4:     lista_pais[i] ← populacao[index]
 5: fimpara
```

```
 6: retorne lista_pais
fimfunção
```

4.5. Passo 4: Crossover para gerar os filhos

Nesse passo foi criado uma função para fazer o crossover dos pais retornados na **função 5**. Para isso foi criada uma função que usa o método de crossover PMX [12] para cada par de pais, com uma taxa de crossover de 0.8.

Função 6: PMX

```
função
        função_6 (pai_s: Lista[str], pai_t: Lista[str], tx_crossover : float) :
Lista[str]
var
        ponto_corte: int
        filho : Lista[str]
        idx : int
inicio
 1: se random() <= tx_crossover então
 2:     ponto_corte ← randint(1, len(pai_s) - 2)
 3:     filho ← cópia de pai_s
 4:     para i de 0 até ponto_corte faça
 5:             idx ← pai_s.index(pai_t[i])
 6:             filho[i] ← filho[idx]
 7:             filho[idx] ← filho[i]
 8:     fimpara
 9:     retorne filho
10: fimse
11: retorne pai_s
fimfunção
```

Função 7: Crossover dos pais

```
função
        função_7 (pais: Lista[lista], tx_crossover : float) : Lista[lista]
var
        lista_filhos : Lista[lista]
        n_pais : int
```

```
        filho_1 : Lista[str]
        filho_2 : Lista[str]
inicio
 1: lista_filhos ← lista de len(pais) listas preenchidas com None
 2: n_pais ← len(pais)
 3: para i de 0 até n_pais passo 2 faça
 4:     filho_1 ← função_6(pais[i], pais[i + 1], tx_crossover)
 5:     filho_2 ← função_6(pais[i + 1], pais[i], tx_crossover)
 6:     lista_filhos[i] ← filho_1
 7:     lista_filhos[i + 1] ← filho_2
 8: fimpara
 9: retorne lista_filhos
fimfunção
```

4.6. Passo 5: Mutação dos filhos

No quinto passo foi criada uma função para realizar a mutação dos filhos retornados na **função 7**. A mutação desenvolvida consiste basicamente em trocar dois pontos de posição em cada filho, usando uma taxa de mutação de 0.01.

Função 8: Mutação de um filho

```
função
        função_8 (ind: Lista[str], tx_mutacao : float) : Lista[str]
var
        index_1 : int
        index_2 : int
inicio
 1: se random() <= tx_mutacao então
 2:     index_1 ← randint(0, len(ind) - 1)
 3:     index_2 ← randint(0, len(ind) - 1)
 4:     ind[index_1] ← ind[index_2]
 5:     ind[index_2] ← ind[index_1]
 6: fimse
 7: retorne ind
fimfunção
```

Função 9: Mutação dos filhos

```
função
        função_9 (filhos: Lista[lista], tx_mutacao : float) : Lista[lista]
var
        função sem variáveis
inicio
 1: para i, individuo em enumerate(filhos) faça
 2:     filhos[i] ← função_8(individuo, tx_mutacao)
 6: fimpara
 7: retorne filhos
fimfunção
```

4.7. Passo 6: Cálculo da aptidão dos filhos

Nesse passo o cálculo de aptidão dos filhos é feito usando as funções **2** e **3** criadas no passo 2.

4.8. Passo 7: Seleção dos sobreviventes para substituição geracional

No sétimo passo é feita a seleção dos sobreviventes para a substituição geracional da população. Para isso, foi criada uma função que faz a substituição geracional com elitismo ou sem elitismo. Utilizando o elitismo, o pior filho é substituído pelo melhor pai, e sem elitismo, a próxima geração será formada pela lista de filhos retornada na **função 9** e pela aptidão dos filhos calculada no **passo 6**.

Função 10: Substituição geracional

```
função
        função_10 (pop : Lista[lista], apt_pop : Lista[float], filhos :
Lista[lista], apt_filhos : Lista[float], elitismo : bool) : Lista[lista] e
Lista[float]
var
        index_melhor_pai : int
        index_pior_filho : int
inicio
 1: se elitismo == True então
 2:     index_1 ← apt_pop.index(min(apt_pop))
 3:     index_2 ← apt_filhos.index(max(apt_filhos))
 4:     filhos[index_pior_filho] ← pop[index_melhor_pai]
 5:     apt_filhos[index_pior_filho] ← apt[index_melhor_pai]
```

6: **fimse**

7: **retorne** filhos, apt_filhos

fimfunção

5. Experimentos

Esta seção apresenta experimentos realizados com o algoritmo desenvolvido, detalhando quais foram as entradas, os resultados e fazendo uma análise dos resultados obtidos.

5.1. Entradas

Foram realizados 4 experimentos utilizando 2 instâncias da base de dados de entrada para o TSP [13]: 2 experimentos com 29 cidades e 2 experimentos com 38 cidades.

Os experimentos foram feitos com a finalidade de comparar as soluções obtidas usando a substituição geracional com elitismo e sem elitismo. Nos quatro testes foram usados os seguintes parâmetros:

Tam. População	Qtde. Gerações	Taxa de Crossover	Taxa de Mutação
Qtde. cidades * 4	5000	0.8	0.01

5.2. Resultados

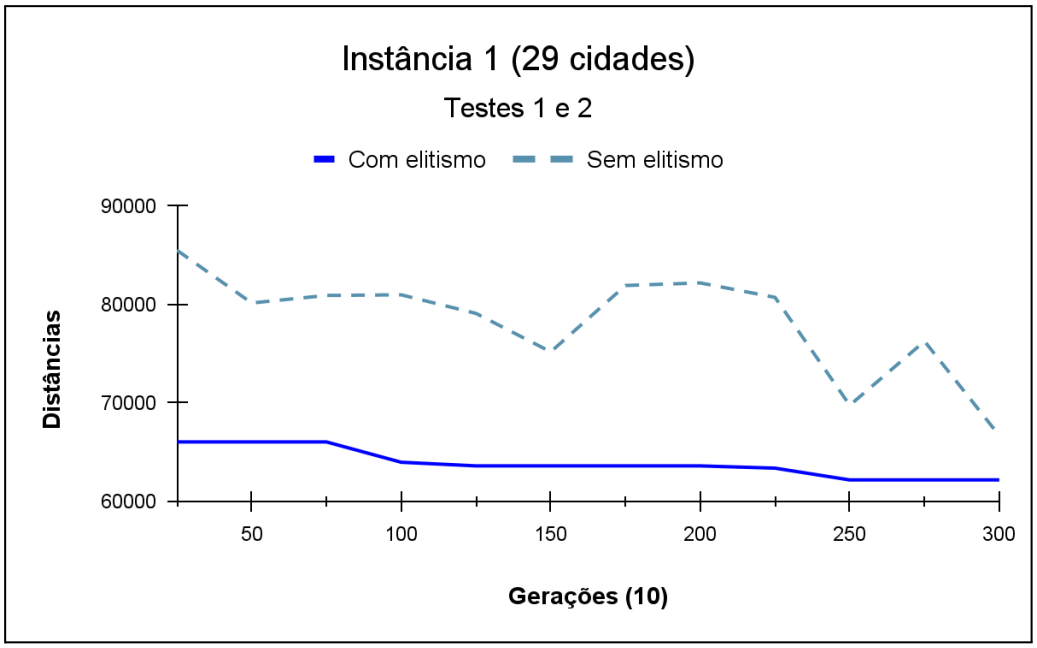
Os resultados obtidos nos mostram a solução ótima obtida em cada teste com e sem elitismo após 3000 gerações, juntamente com a geração em que a solução ótima é encontrada.

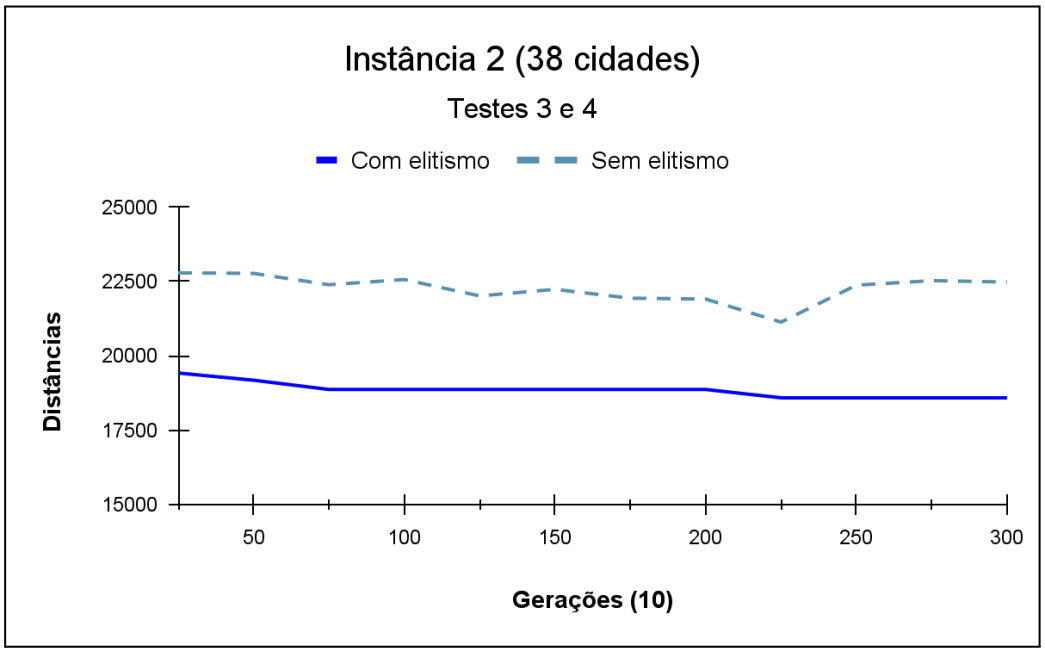
T. nº	Problema	Elitismo	Melhor solução	Geração da melhor solução
1.	wi29	Sim	62181.19436285451	2500
2.	wi29	Não	66677.49862328813	3000
3.	dj38	Sim	18589.150918836396	2250
4.	dj38	Não	21134.23330182988	2250

5.3. Análise de resultados

Com base nos resultados dos experimentos realizados, podemos observar que os algoritmos genéticos que utilizam a seleção

de sobreviventes por elitismo encontram uma solução ótima menor do que aqueles que não usam, além de encontrar a melhor solução com uma menor quantidade de gerações.





6. Resultados

Ao utilizarmos a seleção dos sobreviventes com elitismo, a melhor solução que obtemos tende a ser menor do que a melhor solução obtida sem usar elitismo, além de encontrá-la mais rápido. Além disso, a substituição geracional sem elitismo apresenta uma maior variação da melhor solução de cada população à medida que as gerações passam, diferentemente da substituição geracional com elitismo, que apresenta uma variação menor.

7. Conclusão

O objetivo deste trabalho era desenvolver um algoritmo capaz de calcular a rota de menor custo, de forma que otimize os trajetos dos drones para que eles realizem suas entregas mais eficientemente. Foi implementado um Algoritmo Genético (AG) que consegue encontrar uma boa solução em tempo polinomial, diferentemente do método de força-bruta, que tem uma complexidade de tempo fatorial. Entretanto, pelo fato do AG ser uma metaheurística, ele não garante que a solução obtida será ótima, apenas que será encontrada uma solução próxima da ótima rapidamente.

Portanto, a partir do conhecimento acerca dos problemas NP-Difíceis, metaheurísticas e dos experimentos realizados neste trabalho, concluímos que utilizar o algoritmo genético com seleção dos sobreviventes por elitismo nos dá uma solução melhor do que usar o AG sem elitismo, além de conseguirmos chegar numa solução o mais próximo possível da ótima (ou talvez a ótima) mais rapidamente. Contudo, ao usar o algoritmo genético, nós estamos abrindo mão da garantia de obtermos a melhor solução, o que não acontece no método de força-bruta.

Referências Bibliográficas

[1] CORMEN, Thomas. **Algoritmos - Teoria e Prática**. 3.ed. Barueri: Grupo GEN, 2012.

[2] FEOFILOFF, Paulo. Análise de Algoritmos. **IME-USP**. São Paulo, 2015. Disponível em: https://www.ime.usp.br/~pf/analise_de_algoritmos/. Acesso em 11 jan. 2023.

[3] BRUNET, João. Análise Assintótica. **Joaoarthurbm**. Campina Grande, 2019. Disponível em: https://joaoarthurbm.github.io/eda/posts/analise-assintotica/. Acesso em 12 jan. 2023.

[4] OLIVETE, Celso. Complexidade computacional - classes de problemas. FCT Unesp. Presidente Prudente, [2020?]. Disponível em:

http://www2.fct.unesp.br/docentes/dmec/olivete/tc/arquivos/Aula11.pdf. Acesso em 13 jan. 2023.

[5] DIFERENÇA entre NP Difícil e NP Completo. **Acervo Lima**. c2022. Disponível em: https://acervolima.com/diferenca-entre-np-dificil-e-np-completo/. Acesso em: 13 jan. 2023.

[6] META-HEURÍSTICA. In: WIKIPÉDIA, a enciclopédia livre. Flórida: Wikimedia Foundation, 2022. Disponível em: https://pt.wikipedia.org/wiki/Meta-heur%C3%ADstica. Acesso em: 13 mar. 2023.

[7] ZANCHETTIN, Cleber. Algoritmos Genéticos. CIn UFPE. Recife, 2015. Disponível em: https://www.cin.ufpe.br/~cz/aulas/Algoritmos%20Gen%C3%A9ticos.pdf. Acesso em: 21 mar. 2023.

[8] BARBOSA, C. E. M. **Algoritmos bio-inspirados para solução de problemas de otimização**. Dissertação (Mestrado em Ciência da Computação) - Centro de Informática, Universidade Federal de Pernambuco. Recife, p. 194. 2017. Disponível em: https://repositorio.ufpe.br/handle/123456789/25229. Acesso em: 21 mar. 2023.

[9] KHAN, Fozia Hanif et al. Solving TSP problem by using genetic algorithm. **International Journal of Basic & Applied Sciences**, v. 9, n. 10, p. 79-88, 2009. Disponível em: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=506ce8b5a354a112a3a92c4ff08626db4142aaad. Acesso em: 21 mar. 2023.

[10] GUEDES, A. da C. B.; FIGUEIREDO LEITE, J. N.; ALOISE, D. J. Um algoritmo genético com infecção viral para o problema do caixeiro viajante. Revista PublICa, [S. l.], v. 1, n. 1, 2009. Disponível em: https://periodicos.ufrn.br/publica/article/view/125. Acesso em: 21 mar. 2023.

[11] CARVALHO, M.B.; YAMAKAMI, A.. Meta-heurística Híbrida de Sistema de Colônia de Formigas e Algoritmo Genético para o Problema do Caixeiro

Viajante. **Trends in Computational and Applied Mathematics**, [S.l.], v. 9, n. 1, p. 31-40, june 2008. Disponível em: https://tema.sbmac.org.br/tema/article/view/178. Acesso em: 21 mar. 2023.

[12] Goldberg,D.E., and R. Lingle. “Alleles, Loci, and the Traveling SalesmanProblem.” Proceedings of the First International Conference on GeneticAlgorithms and TheirApplication, edited by Grefenstette J., Lawrence Erlbaum Associates, Hillsdale, NJ,1985, pp. 154-159.

[13] UNIVERSITY OF WATERLOO. **National Traveling Salesman Problem**, 2022. Disponível em: https://www.math.uwaterloo.ca/tsp/world/countries.html. Acesso em: 21 mar. 2023