UNIVERSIDADE ESTADUAL DE MARINGÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA

Modelagem e Otimização Algorítmica: Algoritmo Genético Híbrido para o Problema do Caixeiro Viajante

3a Avaliação

Relatório

Alunos:
Renan Augusto Leonel ra: 115138
José Rafael Silva Hermoso ra: 112685

1 Introdução …….

2 Descrição do Problema ……

3 Descrição do Algoritmo ……

4 Calibrando parâmetros ……..

5 Detalhes de implementação……..

6 Resultado ……..

7 Conclusão…….

8 Referências …….

1. Introdução

Este relatório tem como objetivo apresentar os resultados obtidos na implementação de um algoritmo baseado na meta-heurística “Algoritmos Genéticos” combinado com Busca Local para o problema do Caixeiro Viajante, utilizando os operadores de cruzamento “Position based crossover (POS ou BASEDPOS)” e “Cycle crossover (CX)”. Para isto, optamos por utilizar a linguagem de programação C++.

2. Descrição do Problema

2.1 Problema do caixeiro viajante

O problema do Caixeiro Viajante (PCV) é um problema bastante conhecido na computação, devido a sua classe de otimização NP-difícil, e consiste basicamente em determinar o menor caminho possível para percorrer uma série de pontos de forma que cada ponto seja visitado apenas uma vez, e volte ao ponto de origem. Dada a sua complexidade, utilizaremos um algoritmo meta heurístico para nos auxiliar a chegar em uma solução convincente, pois estes algoritmos nos fornecerão uma solução aceitável de forma rápida,mas não garantindo a solução ótima para o problema em questão.

2.2 Definição de algoritmo genético padrão e híbrido

Um algoritmo genético (AG) é um algoritmo probabilístico inspirado na evolução natural das espécies e na genética. É uma técnica que envolve uma busca adaptativa baseada na sobrevivência dos indivíduos mais aptos da população, juntamente com a reprodução, aplicado para problemas de otimização. Para esse trabalho abordaremos o PCV com o objetivo de encontrar soluções que se aproximem do seu valor ótimo para um conjunto de testes.
Um AG é constituído dos elementos básicos a seguir:

- Cromossomo: um indivíduo
- Gene: um elemento do cromossomo
- População: conjunto de cromossomos
- Crossover (cruzamento): cruzamento dos cromossomos pais para gerar novos cromossomos filhos
- Seleção: seleciona os cromossomos para o crossover
- Mutação: modificação arbitrária de uma pequena parte do cromossomo

Para o PCV, os cromossomos representam uma solução (ciclo hamiltoniano), um gene representa uma parte deste ciclo, a população representa um conjunto de soluções que são os cromossomos e a mutação se dá pela troca aleatória de genes de uma solução. Para selecionarmos os indivíduos, devemos abstrair para uma roleta de probabilidades, simulando uma escolha onde os melhores indivíduos têm

mais chances de serem escolhidos. A seguir, a reprodução é abstraída para um algoritmo que faz o cruzamento de dois indivíduos pré-selecionados. Além desses elementos, também temos a geração de uma população inicial, o critério de parada, o critério de avaliação da população, o critério para ocorrer uma mutação e a forma de realizar a manutenção da população.

```
Gerar população inicial
enquanto critério-de-parada faça:
    avaliação da qualidade dos cromossomos da população
    seleção para o cruzamento
    cruzamento
    mutação
    atualização da população
```

Pseudocódigo de um algoritmo genético padrão.

Agora que temos a base para o nosso algoritmo genético padrão definido, vamos adicionar a busca local, que será feita após a mutação nos filhos resultantes do cruzamento. Com isso temos a definição do algoritmo genético híbrido.

```
Gerar população inicial
enquanto critério-de-parada faça:
    avaliação da qualidade dos cromossomos da população
    seleção para o cruzamento
    cruzamento
    mutação
    buscalocal
    atualização da população
```

Pseudocódigo de um algoritmo genético híbrido.

2.3. Variáveis do algoritmo genético híbrido.

Foi visto acima, que o AG possui vários elementos e muitos deles são parâmetros variáveis, que devem ser ajustados de acordo com o problema. Para o caso deste trabalho as variáveis serão: Critério de parada, tamanho da população inicial e taxa de mutação.
Além dos parâmetros variáveis, temos as estratégias a serem utilizadas para realizar a criação da população inicial, a avaliação dos indivíduos, o cruzamento a ser utilizado, o tipo de busca local e a forma de manutenção da população.
As estratégias utilizadas são:

- gerar uma população inicial de forma aleatória;
- Avaliar os indivíduos utilizando a função fitness, que gera uma porcentagem proporcional a qualidade da sua solução, tal qualidade que

para o PCV pode ser entendida como o custo do caminho analisado (menor o custo, maior a qualidade).
- Selecionar os indivíduos através da roleta com com seus respectivos fitness;
- Utilizar os cruzamentos com os cromossomos selecionados, busca local com 2-opt e best improvement e a manutenção da população baseada no Steady Stated, substituindo os indivíduos menos aptos pelos novos cromossomos filhos, caso sejam mais aptos.

3. Descrição do Algoritmo

Primeiramente, precisamos definir os operadores de cruzamento e busca local utilizados nesta implementação. Optamos por escolher os operadores *POS (Position based Crossover)* e *CX (Cycle crossover)* e a busca local com 2opt.

3.1 POS: Consiste em selecionar um conjunto aleatório de genes de uma solução pai e recolocá-las na outra solução pai, preservando a posição dos genes escolhidos aleatoriamente do primeiro pai, assim gerando um cromossomo filho com característica dos dois cromossomos pai. Utilizando a nomenclatura de filho1 e filho2 e pai1 e pai2, começamos selecionando o conjunto de posições aleatórias e copiando o pai1 para o filho1 e pai2 para o filho2. Em seguida, para todas as posições selecionadas, chamadas de i, fazemos com que o filho1 na posição i receba o pai2 na posição i, achamos o índice do gene do filho1 na posição i no cromossomo do pai1 e salvamos sua posição e por fim fazemos o filho1 na posição salva receber o pai1 na posição i. Vale ressaltar que o mesmo deve ser feito para o filho2, mas com o outro pai.

3.2 CX: Consiste em gerar descendentes através do preenchimento de genes proveniente dos pais, ou seja, cada gene de cada pai ocupa exatamente a mesma posição no cromossomo do filho, gerando um filho com as características dos dois pais. Começamos com dois filhos inicializados como vazio, nomeados como filho1 e filho2, e os pais, pai1 e pai2. Primeiramente copiamos o pai2 para o filho1 e utilizando uma variável idx, para guardar o índice dos genes, que começa em 0, colocamos o gene na posição idx do pai1 no filho1. Em seguida buscamos o índice do gene localizado na posição idx do filho1 no cromossomo do pai2 e atualizamos o valor de idx. Este processo é feito até que o cromossomo filho1 seja montado por completo. Vale ressaltar que o mesmo deve ser feito para o filho2, mas invertendo o pai, a cópia e a busca do idx deve ser feita do pai1.

3.3 2-opt: Iniciamos a partir de uma solução já conhecida, nesse caso um cromossomo da população e, fixamos um vértice inicial e um segundo vértice, em seguida efetuamos a troca desses dois vértices e colocamos o caminho entre eles de forma inversa, o que gera um novo caminho. Para o vértice inicial podemos

atribuir qualquer vértice i pertencente ao grafo que representa o caminho e para o segundo vértice atribuímos o nó na posição i+1 desse mesmo grafo. Caso o novo caminho seja melhor que o caminho inicial, esta passa a ser a nova solução para o problema, e repetimos este método até que nenhuma melhoria possa ser aplicada à solução. Os parâmetros i e j devem sempre ser menores que o tamanho total do grafo.

Em seguida vamos definir como a avaliação e a seleção serão realizadas.

3.4 Avaliação: Calculamos a distância de cada solução e calculando seu fitness, fazendo o inverso do valor calculado, sendo assim, a solução com o maior fitness terá o menor custo (distância do caminho).

3.5 Seleção: Com o fitness calculado, calculamos a porcentagem de cada solução escolhida, utilizando seu fitness e o valor total dos fitness da geração atual. Com as porcentagens calculadas, obtemos os intervalos correspondentes a cada solução. Com isso, sorteamos um número aleatório de zero a cem e pegamos a solução em que esse número se encaixa no intervalo.

Por fim temos as tarefas mais simples, gerar a população inicial, a mutação e a manutenção da população.

3.6 Geração da população inicial: Utilizando parâmetro que indica a quantidade de indivíduos na população, geramos caminhos construídos de forma aleatória, que serão nossos cromossomos, e os inserimos na população inicial.

3.7 Mutação: A mutação ocorre de uma maneira simples. Primeiramente, escolhemos um número de forma aleatória, se ele for menor ou igual à taxa de mutação estabelecida escolhemos dois pontos de forma aleatória no cromossomo que sofrerá a mutação e fazemos a troca entre eles, trocando esses 2 genes um com o outro.

3.8 Manutenção da população: Utilizando a estratégia Steady Stated, procuramos na população atual o indivíduo com o menor fitness e caso o filho gerado pelo cruzamento tenha um fitness maior, ou seja, a distância da sua solução seja menor, retiramos o menor e colocamos o filho em seu lugar. Esse método faz com que o tamanho da população seja constante durante a execução e que ao longo do tempo as escolhas através da roleta sejam cada vez melhores, visto que os piores indivíduos vão dando espaço para os melhores. Outro ponto que foi levado em consideração foi o tempo de execução, que para outras estratégias como a populacional, seria muito maior, visto que uma nova geração teria que ser feita do zero a cada iteração do algoritmo, fazendo com que a busca local seja executa muitas vezes gerando um gargalo de tempo, o que não ocorre com o Steady Stated, onde a busca local é reduzida a até duas execuções por geração.

4. Calibrando parâmetros

Como foi citado anteriormente, possuímos parâmetros variáveis, e para o algoritmo utilizado utilizaremos a população, a taxa de mutação , um fator de estagnação para as soluções e a parada caso um número pré determinado de gerações seja atingido. Começando com o fator de mutação, sabemos que uma taxa de 1% a 5% são valores aceitáveis para fazer com que a solução tenha chances de escapar de casos de melhor local, por exemplo. Analisando esse parâmetro, foi notado que, quando muito baixo, 1% ou 2%, a estagnação acontecia por muitas gerações, fazendo com que a solução ficasse no mesmo valor por muito tempo. Subindo o valor de mutação para a casa dos 5%, valor utilizado nos testes principais, esse problema diminui consideravelmente, porém ainda continua acontecendo. O problema da estagnação é inevitável, uma tentativa de burlar este problema é subir a taxa de mutação, porém isso faz com bons indivíduos possam ser destruídos, visto que a chance deles serem mutados passa a ser alta.

Outro fator que caminha lado a lado com a mutação é o tamanho da população, que também é um parâmetro variável. Caso a população seja muito pequena, problemas com a variabilidade das soluções vão ocorrer, fora que a mutação vai ter um efeito muito mais forte, visto que a população possuirá poucos indivíduos. Por outro lado, se a população for muito grande, a taxa de mutação tem que acompanhar, pois se for muito pequena o problema da estagnação sofrerá uma piora e quando houver um decaimento do valor da solução, será muito pequeno.

A solução para isso é balancear para esse algoritmo onde uma população abaixo de 15 pode ser considerada pequena, fazendo com que demore muitas gerações para que o custo da solução diminua, e para valores maiores, como 30 ou superiores, afetam no desempenho, visto que a estratégia usada para a busca local é o best improvement, o que para os casos maiores demora um tempo consideravelmente grande para ser executado.

Sendo assim, foi escolhido o tamanho de 20 indivíduos por população. Também temos o fator de estagnação, que verifica se a solução foi repetida até x vezes, caso sim, a execução é interrompida, caso não, continuamos a execução. Vale ressaltar que esse fator sofre um reset quando a solução é melhorada. O valor x citado é bem volátil, por exemplo, para os casos pequenos ele pode simplesmente não existir, visto que a execução é muito rápida, o que faz com que mesmo com muita estagnação o custo da solução acaba sendo melhorado em um curto espaço de tempo. Por outro lado , quanto maior a entrada se torna, o valor de x deve ser ajustado, para casos como pla33810 e pla85900, ele deve ser bem baixo, como 10 ou até mesmo 5, visto que quanto maior for o x mais execuções vão acontecer, e esses casos se tornam extremamente lentos devido ao seu tamanho.

Por fim, temos a parada por número de gerações, que também é volátil e depende muito da entrada que vai ser executada, podendo executar até duas mil gerações para o menor caso, o pr1002, e apenas vinte vezes para casos como pla85900.
Vale ressaltar que o critério de parada utiliza o número de gerações e a estagnação, o que ocorrer primeiro faz com que o algoritmo pare.

5. Detalhes de implementação.

- Primeiramente, usamos um vetor para guardar a entrada, os valores de x e y, e usamos esse vetor apenas como referência, pois todas as operações são feitas com vetores de índices e não com as posições x e y.

- Começamos o algoritmo gerador para uma população de caminhos aleatórios, tais caminhos são gerados através do embaralhamento de um vetor com os índices, de 0 à len(entrada) -1.

- A implementação da roleta foi adaptada, a ideia das faixas de porcentagem foi mantida, porém para simplificar o sorteio, a porcentagem foi arredondada para o valor inteiro mais próximo e um vetor foi criado para guardar o índice do cromossomo e suas repetições, tais repetições que representam a sua faixa de porcentagem. Por exemplo, se o indivíduo que possui 29% de chance de ser escolhido, arredondamos para 30 e distribuímos 30 k no vetor, fazendo com que se o valor sorteado for de 1 a 30, o indivíduo k vai ser selecionado.

- A busca local foi adaptada, passando a utilizar o vetor de índices e não o vetor que contém as informações das posições, como foi feito no trabalho anterior.

- Após o cruzamento, temos dois cromossomos filhos, porém aplicamos um elitismo e escolhemos o melhor filho para executar a busca local e fazer sua inserção na população. Tal procedimento faz com o tempo de execução seja diminuído, uma vez que a busca local só será executada uma vez por geração.

- Para o cruzamento POSBASED, é utilizado um conjunto de posições aleatórias, que para evitar gasto computacional extra para gerar um conjunto e verificar se todos os elementos são diferentes, foi ajustado para dez posições, sendo elas já pré-escolhidas arbitrariamente.

- Para os maiores casos de teste, a estratégia de busca local foi alterada para first improvement, buscando reduzir o gargalo de tempo gerado pela busca local 2opt.

- Os algoritmos vão estar compilados, bastando apenas rodar o executável com o nome do algoritmo desejado. Vale ressaltar que as variáveis como taxa de mutação, população e critério de parada podem ser alteradas no começo de cada código, porém uma nova compilação com o g++ será necessária. Leia o arquivo readme para visualizar como os parâmetros estão para localizá-los.

6. Resultados

6.1. Resultados da execução para os casos de teste.

Os resultados obtidos utilizam a chance de 5% de mutação, como já foi citado, e uma adaptação de parâmetros, começando sem o critério de parada por estagnação para o pr1002 e fnl4461 e para um número alto de gerações, como 10000 e 2000.
Para os outros casos, o número de gerações foi diminuído, atingindo até 100 gerações e o critério de estagnação foi adicionado para parar a execução. Vale ressaltar que o melhoramento 2opt é executado uma vez a cada geração, o que faz com que os casos grandes levem um tempo alto para cada geração, tornando inviável a execução por muitas gerações.

Cruzamento CX			
entrada	**custo**	**taxa de mutação**	**% erro**
brd14051	517.862,75	5%	10,43%
d15112	1.753.887,13	5%	11,49%
d18512	716.859,13	5%	11,10%
fnl4461	196.474,95	5%	7,62%
pla7397	25.394.526,00	5%	9,17%
pla33810	79.394.312,00	5%	20,21%
pla85900	250.672.288,00	5%	76,06%
pr1002	264.936,25	5%	2,27%

Tabela 1: Resultados para o cruzamento CX.

Cruzamento BASED			
entrada	**custo**	**taxa de mutação**	**% erro**
brd14051	519.623,28	5%	10,81%

d15112	1.738.451,25	5%	10,51%
d18512	709.314,38	5%	9,93%
fnl4461	197.476,17	5%	8,17%
pla7397	25.198.134,00	5%	8,33%
pla33810	81.293.560,00	5%	23,08%
pla85900	367.321.504,00	5%	157,98%
pr1002	265.126,41	5%	2,35%

Tabela 2: Resultados para o cruzamento BASED.

6.2 Comparação com o trabalho de algoritmos construtivos e melhorativos.

Começando com o 2opt e levando em consideração que a solução foi gerada pelo algoritmo de construção vizinho mais próximo, e não de forma aleatória como no trabalho atual temos os seguintes resultados.

arquivo de texto	**vizinho mais próximo + 2opt**	**OPT**	**%erro**
	enquanto houver melhoria		
brd14051	500.653,09	468.942,40	6,76%
d15112	1.677.157,00	1.573.084,00	6,62%
d18512	686.937,63	645.238,00	6,46%
fnl4461	194.486,95	182.566,00	6,53%
pla7397	24.673.252,00	23.260.728,00	6,07%
pla33810	69.983.888,00	66.048.945,00	5,96%
pla85900	149.926.336,00	142.382.641,00	5,30%
pr1002	274.950,88	259.045,00	6,14%

Tabela 3. Resultados obtidos utilizando as heurísticas do vizinho mais próximo + 2opt enquanto houver melhoria.

Comparando os resultados, podemos perceber que o vizinho mais próximo junto com o 2opt acaba sendo melhor que o algoritmo genético, porém, os resultados do algoritmo genético, algumas vezes, não se afasta tanto, o que nos dá um forte indício de que se for investido mais tempo durante a execução do algoritmo genético, sua solução pode ser cada vez melhor. Tal teoria se mostra promissora,

visto que para o teste pr1002, que por ser pequeno, foi possível executar muitas iterações em pouco tempo, nos dando um resultado melhor que o vizinho mais próximo com o 2opt. Ainda sobre tempo de execução, o vizinho mais próximo com o 2opt é executado em em tempo muito menor que o algoritmo genético, visto que para o primeiro o melhoramento é feito uma vez, e para o algoritmo genético desenvolvido ele é executado uma vez por geração, gerando um gargalo de tempo. Tal gargalo acaba se pagando, visto que é feita a troca de tempo de execução por uma boa melhoria na solução. Uma melhoria muito melhor do que o cruzamento e a mutação poderia oferecer.

Também temos o 3opt com sua estratégia de construção de solução aleatória com os seguintes resultados.

arquivo de texto	**vizinho aleatório + 3opt**	**solução ótima**	**gap%**
	30min		
brd14051	529763.687500	468942.4	12,97%
d15112	1769015.000000	1573084	12,46%
d18512	723487.750000	645238	12,13%
fnl4461	205856.703125	182566	12,76%
pla7397	26267562.000000	23260728	12,93%
pla33810	79402624.000000	66048945	20,22%
pla85900	3227562752.000000	142382641	2166,82%
pr1002	282014.468750	259045	8,87%

Tabela 4. Resultados obtidos utilizando as heurísticas vizinho aleatório + 3opt com 30 minutos de execução limite

Já para o 3opt o cenário é outro, nesse caso o algoritmo genético híbrido se mostrou melhor que o vizinho aleatório e o 3opt, reforçando a ideia de que o algoritmo genético pode gerar uma solução muito boa, mas ainda sim sem garantia. Um fator a ser levado em conta é que como o 3opt foi executado para soluções aleatórias, assim como o algoritmo genético, não desempenhando 100%, visto que, como ele faz mais trocas que o 2opt, ele deveria ser melhor.

6.2 Comparação entre os cruzamentos.

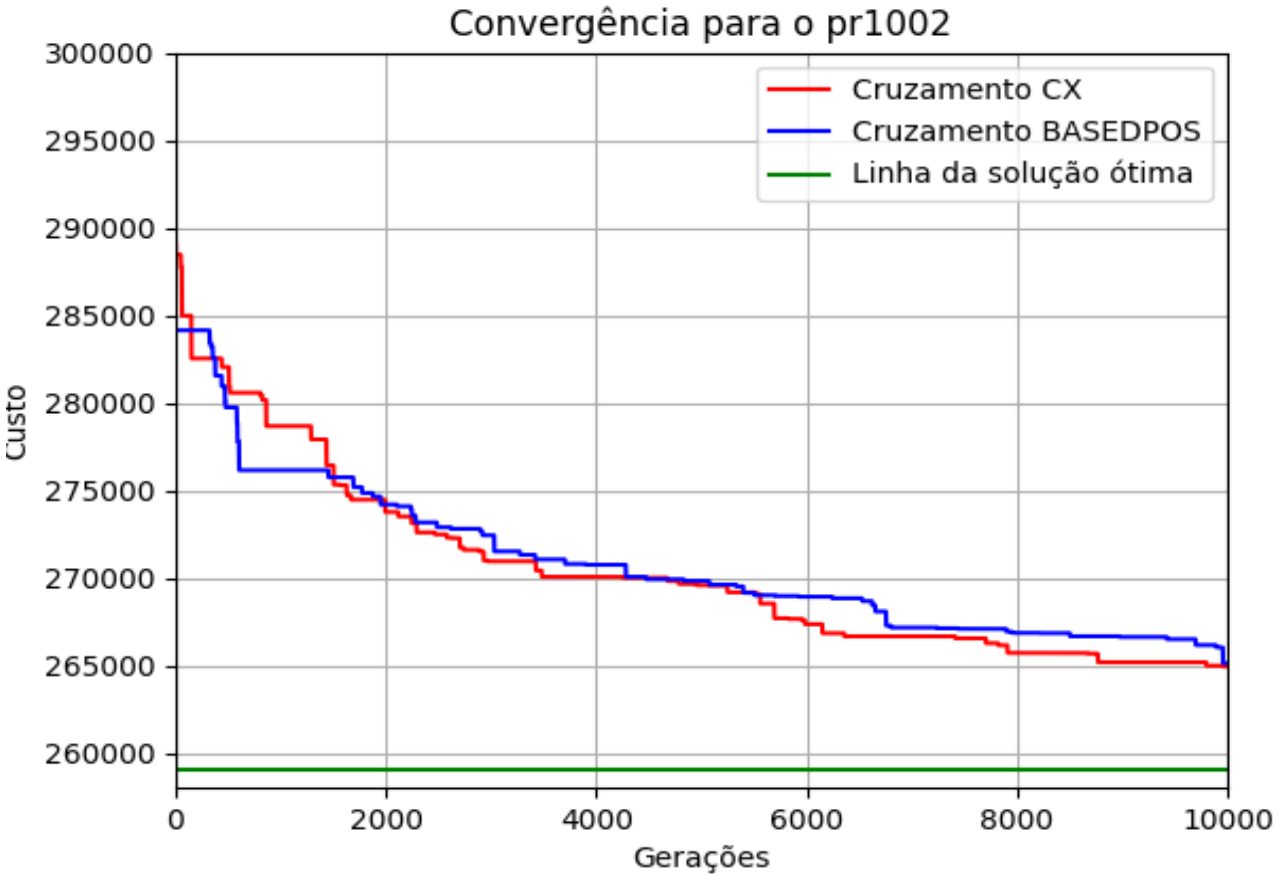

Imagem 1. Gráfico de convergência para o pr1002.

Imagem 2. Gráfico de convergência para o pla85900.

Como podemos observar nas imagens acima, o cruzamento BASEDPOS consegue realizar o seu propósito, gerando uma solução melhor a partir de outras 2, porém, o cruzamento CX se destaca, gerando uma solução melhor que o primeiro. Tal fato ocorre, pois o BASEDPOS é bem mais simples que o CX, fazendo trocas especificas, visto que utiliza um conjunto de posições pre-definidos para o cruzamento. Já o CX, utiliza o principio de buscar o índice do valor do filho no vetor do pai que também é utilizado pelo BASEDPOS, porém, tal procedimento ocorre para todo o cromossomo do pai, não utilizando posicoes pre-definidas.

7. Conclusão

Este trabalho nos possibilitou implementar uma meta heurística para conseguir chegar em uma solução para o problema do caixeiro viajante. Tal implementação se refere ao algoritmo genético com melhoramento com busca local, sendo denominado algoritmo genético híbrido.

Neste trabalho, a implementação foi simples, visto que o algoritmo genético é um conjunto de procedimentos simples, exceto pela roleta que foi um pequeno desafio adaptar como foi descrito na seção de detalhes de implementação.

Ao longo do desenvolvimento o maior empecilho foi o tempo, visto que a implementação do algoritmo proposto neste trabalho envolve a execução da busca

local 2opt, que consome um tempo proporcional ao tamanho da entrada. Para entradas pequenas como o pr1002 conseguimos nos aproximar bastante do seu valor ótimo, mais que no trabalho anterior, porém para os outros casos, chegamos perto do valor ótimo em alguns casos, mas mesmo assim os resultados foram majoritariamente piores que no trabalho anterior. Porém, como já citado, o algoritmo genético com busca local, se for executado com o intuito de chegar próximo ao valor ótimo e não só obter uma solução boa, ele pode vir a cumprir seu papel, já que é uma meta heurística e não possui garantia de gerar um bom resultado, ao custo de tempo de execução. Para se ter uma base, uma única execução para o pla85900 do 2opt com best improvement demorou cerca de 30 minutos no trabalho anterior e teve uma boa aproximação do seu valor ótimo. Tendo isso em vista, para executarmos o nosso algoritmo genético para este caso utilizando uma população de 20 indivíduos, seria executado 20 2opt demorando cerca de 600 minutos, isso sem contar o tempo para as outras computações, como o cruzamento. Esse tempo pode se agravar ainda mais caso seja escolhido o 3opt como algoritmo de busca local para o algoritmo genético.

Um outro ponto importante a ressaltar, é a diferença de resultados entre os dois cruzamentos utilizados, um teste com a entrada att48 foi realizado e nela foi visto que o cruzamento BASEDPOS funciona bem para instâncias pequenas, se igualando ao CX, porém, para as instâncias de entrada selecionadas para este trabalho, podemos notar logo na primeira que o CX é superior, o que é confirmado pela diferença apresentada na maior entrada. Mas, o CX ser superior não suprime o fato de que ambos os cruzamentos convergem para a solução ótima.

Por fim podemos concluir que um algoritmo genético precisa de tempo para que sua característica evolutiva se destaque e gere soluções cada vez melhores, tais soluções que podem vir a se aproximar muito da solução ótima para o caso executado. Porém, se a disponibilidade de recursos, como o tempo, for baixa, a busca local combinada com o vizinho mais próximo pode ser uma boa escolha, visto que também gera uma solução boa para o problema.

8. Referências

Material disponibilizado pelo professor na plataforma Moodlep.

Análise de operadores de cruzamento genético aplicados ao problema do Caixeiro Viajante. **Samuel Wanberg Lourenço Nery 2017.**