# Relatório

**Nome**: Henrique Andrade Lopes
**Matrícula**: 105459

## Introdução

O problema abordado neste trabalho consiste em implementar 2 métodos heurísticos para resolver uma variante do Problema do Caixeiro Viajante (TSP). O TSP envolve determinar a rota de menor distância total que visita todas as cidades exatamente uma vez e retorna à cidade de origem, dado um conjunto de cidades e as distâncias entre elas, nesse problema começaremos sempre da cidade 1 .

Na variante estudada neste trabalho, chamada de TSPP (Traveling Salesman Problem with Preferences), cada cidade tem a capacidade de indicar quando prefere receber a visita do caixeiro. Algumas cidades preferem ser visitadas no início da rota, enquanto outras preferem ser visitadas no meio ou no final. Caso as preferências não sejam seguidas, são aplicadas multas (penalties) estipuladas por cada cidade. O custo total de uma rota é calculado pela soma do custo de percorrê-la (proporcional à distância total) com o custo das multas.

As duas heurísticas implementadas foram Tabu Search e GRASP. Este relatório conterá os pseudocódigos das duas heurísticas, explicação detalhada de cada uma delas e, por fim, uma comparação dos resultados obtidos por meio de cada heurística.

## Métodos

GRASP:

A heurística GRASP (Greedy Randomized Adaptive Search Procedure) é implementada da seguinte maneira:

1. A função grasp recebe uma rota inicial, tempoMax que define o tempo máximo de execução e alpha, que controla o nível de aleatoriedade da heurística.

2. É criada uma novaRota vazia e adicionada a cidade 1, pois sempre começaremos a partir dessa cidade.

3. Em seguida, é criada uma lista chamada RLC (Restricted Candidate List) com todas as cidades disponíveis (ainda não visitadas na rota) ordenadas pelo custo de ir da cidadeAtual até a cidadeX acrescido da multa da cidadeX por ser a i-ésima cidade visitada. A posição 0 da lista será a cidade mais "gulosa", ou seja, com menor custo.

4. O valor de k é definido como a soma de min e alpha multiplicado por (max - min), onde min é o custo da cidade na posição 0 da RLC e max é o custo da última cidade da RLC. Esse valor de k é usado para determinar quais cidades serão consideradas na construção da nova rota.

5. A RLC é atualizada, removendo as cidades cujo custo seja maior que k.

6. Em seguida, uma cidade c é selecionada aleatoriamente da RLC atualizada e é adicionada ao final da novaRota.

7. É realizada uma busca local na novaRota para melhorar a solução.

8. Os passos de 3 a 7 são repetidos até que a novaRota esteja completa, ou seja, todas as cidades tenham sido visitadas.

9. Enquanto ainda houver tempo disponível, o algoritmo repete os passos de 2 a 8, construindo uma nova rota e atualizando a melhor rota encontrada caso seja obtida uma solução com menor custo.

10. Ao final, a função retorna a melhor rota encontrada.

É importante observar que o valor de alpha controla o equilíbrio entre a escolha gulosa (quando alpha é próximo de 0) e a escolha aleatória (quando alpha é próximo de 1). Quanto mais próximo de 0, mais focada na busca por soluções melhores a heurística se torna, enquanto que um valor próximo de 1 introduz maior aleatoriedade na escolha das cidades.

Pseudocódigo da heurística GRASP:

```
grasp(rota, tempo_max, alpha)
    n = tamanho(rota)
    tempo_gasto = 0

    enquanto tempo_gasto < tempo_max:
        visitado[n] = falso
        nova_rota[0] = 1

        para i de 0 até n-1:
            pq = encontraListaCidadesNaoVisitadas()
            k = min + alpha(max - min)
            retira de pq as cidades com custo maior que k
            seleciona cidade "c" pertencente a pq aleatoriamente
            insere "c" em nova_rota
            atualiza visitado[c] = true
            realiza buscaLocal em nova_rota

        se custo_nova_rota < melhor_custo:
            melhor_rota = nova_rota
            melhor_custo = custo_nova_rota

    retorne melhor_rota
```

Tabu Search:

A heurística Tabu Search é implementada da seguinte forma:

1. A função tabuSearch recebe o tempoMax que define o tempo máximo de execução, o tamanho tabu que controla por quantas iteracoes uma mudanca será tabu, o maxIteracoes que contrala quantas iteracoes o algoritmo irá realizar em uma rota e recebe um qtdRotas que será quantas vezes o algoritmo irá gerar uma nova rota.

2. Inicializamos o melhorCusto = INFINITO melhorRota = []

3. O algoritmo comeca gerando uma rota aleatoria (comecando da cidade 1)

4. Definimos uma tabuList[n][n] = 0, sendo n o tamanho da rota. Essa Tabu list ira guardar por quanto tempo um movimento sera tabu, inicializamos com 0 pois no comeco nenhum movimento sera tabu

5. Depois geramos toda a vizinhança da rota (a vizinhança será uma troca 2-opt), se tabuList[i][j] nao for 0 quer dizer que aquele movimento é tabu, entao nao a adicionaremos na vizinhança

6. Essa vizinhança será ordenada pelo custo, a vizinhança com menor custo será a primeira da lista e a de maior sera a ultima

7. Nós realizamos o melhor movimento, ou seja, se a melhor vizinhança é trocar a cidade 4 com a 7 realizamos essa troca, atualizamos o tabuList[4][7] = tabuList[7][4] = tamanhoTabu. Claro, pode acontecer de o melhor vizinho ser pior que a rota atual, mas a atualizamos mesmo assim

8. Se o custo dessa rota for melhor que o da melhorRota atualizamos as variaveis melhorCusto e melhorRota

9. Decrementamos os valores de tabuList que nao sao 0.

10. Voltamos ao passo 5 enquanto nao atingirmos o max iteracoes ou tempoMax

11. Voltamos ao passo 3 enquanto qtdRotas nao for atingida

12. Retornamos a melhor Rota

```
tabusearch(tempoMax, tamanhoTabu, maxIteracoes, qtdRotas):

        melhorCusto = INF; melhorRota=[]

        x:
        rota = geraRotaAleatoria()
        tabulist[n][n]=0

        for(int i=0; i<maxiteracoes; i++)
                vizinhos = geraVizinhosRota(rota, tabulist)
                ordenaVizinhosCusto(vizinhos)

                i = vizinhos[0].i;
                j = vizinhos[0].j

                tabuList[i][j] = tabuList[j][i] = tamanhoTabu
                trocaCidadesLugar(rota, i, j)

                custo = calcCustoRota(rota)

                se custo<melhorCusto
                        atualiza melhorRota e melhorCusto

        qtdRotas-=1

        se qtdRotas > 0 goto x:

        return melhorRota
```

## Comentários sobre otimizaçoes nas heurísticas:

Na heurística GRASP, em instâncias grandes, o processo de realizar uma busca local a cada iteração pode se tornar demorado. Por exemplo, a busca local pode levar cerca de 25 segundos para ser concluída em grafos com aproximadamente 1000 vértices. Portanto, em rotas grandes (com mais de 300 vértices), na heurística GRASP, não realizaremos uma busca local completa. Em vez disso, faremos uma busca local por um determinado período de tempo.

Nos resultados apresentados abaixo, foi considerado um tempo máximo de 0.5 segundos para a busca local na heurística GRASP.

Na heurística Tabu Search, o processo mais demorado é a geração da vizinhança, que tem uma complexidade aproximada de O(n^2) para cada movimento tabu. Durante a implementação, foi adotada a estratégia de gerar apenas uma quantidade limitada de vizinhança a cada iteração, o que permitia ao algoritmo realizar mais trocas tabu, embora com uma vizinhança menor. Essa abordagem ajudou a reduzir o tempo de geração da vizinhança, mas como a qualidade dessa vizinhança era inferior, a diferença nos resultados obtidos não foi significativa. Portanto, optou-se por permitir que o algoritmo gere a vizinhança completa, como mencionado anteriormente.

Uma otimização realizada foi a exclusão de vizinhos que já estão presentes na lista tabu, o que contribui para reduzir a busca por vizinhos. No entanto, essa abordagem também elimina o fator de aspiração do Tabu Search.

Por fim, outra otimização aplicada ao Tabu Search em grafos grandes foi a seguinte: se um vizinho encontrado for melhor do que a melhor rota já encontrada até o momento, não é necessário gerar a vizinhança completa novamente. Isso economiza tempo computacional, uma vez que não é preciso explorar todas as possibilidades.

## Resultados

Os resultados abaixo foram obtidos executando os algoritmos por 15s para cada instância,com excessao do gil262, gr666 e dsj1000 que foram executados por 60, 120 e 300s respectivamente, porém os executei várias vezes durante a fase de teste e coloquei na tabela o melhor resultado obtido.

Realizando uma comparacao visual das duas heuristicas abaixo percebemos que o GRASP obteve resultados levemente melhores em termos de custo que a heurística Tabu Search.

| Instância | Multas | Henrique | |
|---|---|---|---|
| | | GRASP | Tabu Search |
| burma14 | zero | 30 | 30 |
| | cedo | 48 | 48 |
| | mix | 57 | 57 |
| berlin52 | zero | 7542 | 7542 |
| | cedo | 7872 | 7918 |
| | cedo2 | 7987 | 7972 |
| | mix | 7949 | 7937 |
| | mix2 | 8378 | 8296 |
| st70 | zero | 680 | 676 |
| | cedo | 742 | 746 |
| | cedo2 | 825 | 836 |
| | cedo3 | 816 | 849 |
| | mix | 840 | 869 |
| | mix2 | 931 | 956 |
| gil262 | zero | 2578 | 2628 |
| gr666 | zero | 3405 | 3367 |
| dsj1000 | zero | 20370717 | 19983613 |
| gil262 | cedo | 2796 | 2932 |

| | | | |
|---|---|---|---|
| | mix | 3341 | 3379 |
| gr666 | cedo | 5687 | 5373 |
| | mix | 4586 | 5141 |
| dsj1000 | cedo | 21050719 | 22831392 |
| | mix | 21423593 | 22004563 |

## Análise Resultados

Como mencionado anteriormente, a heurística GRASP apresentou resultados melhores, especialmente para grafos maiores. No entanto, é importante considerar também o aspecto do tempo. Durante o processo de implementação e testes das duas heurísticas, observei que o GRASP tende a gerar suas melhores rotas no início do algoritmo, independentemente do tempo de execução - seja 30 segundos ou 30 minutos, os resultados são bastante semelhantes.

Por outro lado, no Tabu Search, quanto mais tempo o algoritmo é executado, melhores são as rotas geradas. Um exemplo disso é a instância "dsj1000 zero". Nesse caso específico, deixei as heurísticas sendo executadas por mais tempo (1 hora) para analisar o que aconteceria. E o Tabu Search superou o GRASP em cerca de 2% para essa instância. Portanto, se houver tempo suficiente disponível para gerar a solução, acredito que o Tabu Search seja a melhor opção.

Isso ocorre porque o Tabu Search explora uma quantidade maior de vizinhos em comparação ao GRASP, o que contribui para a melhoria das soluções encontradas.

## Conclusao

Ambos os algoritmos apresentam resultados satisfatórios quando comparados com as melhores soluções. A escolha entre eles dependerá da instância em questão e do tempo máximo disponível para a execução. Se o problema precisa ser resolvido em um curto período de tempo (por exemplo, 30 segundos), o GRASP seria a opção mais adequada, pois a etapa mais demorada é a busca local, que pode ser ajustada de acordo com o tempo disponível. O GRASP produz boas soluções de forma relativamente rápida, mas geralmente não melhora significativamente a solução ao longo do tempo.

Por outro lado, se houver uma janela de tempo maior disponível (por exemplo, 5 minutos ou mais), o Tabu Search seria mais recomendado, pois ele terá a capacidade de explorar melhor a vizinhança. Com uma janela de tempo de 5 minutos, o Tabu Search pode obter soluções melhores que o GRASP para grafos com menos de 300 vértices. No entanto, ao contrário do GRASP, o tempo de execução do Tabu Search aumenta consideravelmente à medida que o tamanho do grafo aumenta.

Portanto, a escolha entre GRASP e Tabu Search dependerá das restrições de tempo e do tamanho do grafo. O GRASP é mais adequado quando há pouco tempo disponível e para grafos maiores, enquanto o Tabu Search é mais recomendado quando há uma janela de tempo maior e para grafos menores. É importante considerar esses aspectos ao decidir qual algoritmo utilizar para solucionar o problema em questão.