**Trabalho**

**MC548**

**Grupo:**

João Luiz Nogueira Melo Lima – 071269

Leandro G. Conde das Neves – 071459

João Paulo Dejavite Araújo – 073245

**Professor:**

Eduardo C. Xavier

**1s 2011**

1. **Organização e breve descrição dos arquivos código fonte:**

Nosso trabalho foi implementado na linguagem Java utilizando o eclipse como plataforma de desenvolvimento. Nosso projeto possui 5 classes, segue abaixo uma breve descrição de cada uma delas:

**SaveWorld.java:**

Responsável por ler a instância do arquivo, executar o algoritmo (chamar classe Grasp) e imprimir a solução no formato padrão.

**Grasp.java:**

Possui toda a implementação do algoritmo GRASP, como busca local, solução gulosa aleatória e cálculo de vizinhança.

**Station.java:**

Representa a torre descrita no problema, guarda seu número, custo pontos cobertos e sua ponderação (divisão do custo da torre pelo número de pontos coberto por ela, ou seja, quanto menor este valor mais pontos ela cobre com menos custo).

**StationList.java:**

Possui uma lista de Station (torres) e o custo total de todas elas, seria uma representação de uma solução.

**Teste.java:**

Executa teste de acordo com parâmetros previamente setados, utilizada apenas no desenvolvimento do projeto para achar um melhor valor para diversas constantes no nosso algoritmo.

No nosso algoritmo existem diversas constantes e que seus valores vão de acordo com testes realizados antes da entrega do projeto. Segue abaixo uma breve descrição de cada uma delas:

***RANDOM\_LIMIT:***

Dita a aleatoriedade na escolha do elemento da lista restrita de candidatos, representa o número de elementos que podemos escolher, pro exemplo, caso seu valor seja 100 podemos escolher qualquer um dos 100 primeiros elementos da lista restrita de candidatos, esta lista está ordenada de acordo com a ponderação de cada torre.

***MAX\_NEIGHBORS\_REMOVE :***

Dita quantos elementos, ordenados decrescentemente por ponderação, podemos retirar, um por vez, da solução atual durante a busca local. Esta variável busca acelerar, retirando menos elementos da solução atual por iteração, e otimizar, retirando apenas os piores elementos da solução atual, a busca local.

***MAX\_NEIGHBORS\_ADD:***

Dita quantos elementos, ordenados crescentemente por ponderação, podemos adicionar , um por vez, na solução atual durante a busca local. Esta variável busca acelerar, adicionando menos vizinhos por iteração, e otimizar, adicionando apenas os melhores elementos fora solução atual, a busca local.

***LOCAL\_TIME\_LIMIT :***

O limite de tempo de execução para cada busca local. Esta variável busca acelerar a busca local, e forçar um maior número execuções da busca local com o objetivo de escolhermos mais soluções aleatórias iniciais.

1. **Heurísticas utilizadas:**

Utilizamos como base o algoritmo GRASP (Greedy Randomized Adaptive Search Procedures) com algumas modificações. Na nossa solução utilizamos as seguintes heurísticas:

* **Escolha gulosa aleatória:**

Colocamos um valor limite (*RANDOM\_LIMIT*) para pegarmos sempre os primeiros candidatos da lista ordenados crescentemente por ponderação visando a geração de soluções iniciais melhores, mas ainda aleatórias.

* **Vizinhança:**

Colocamos limites para troca de vizinhos, tanto para escolher quais elementos das soluções tiraríamos (*MAX\_NEIGHBORS\_REMOVE*) quanto para escolher quais elementos fora da solução incluiríamos (*MAX\_NEIGHBORS\_ADD*), obviamente seria uma por vez. Além disso, ordenamos os elementos da solução, por ordem decrescente de ponderação, e os fora da solução, por ordem crescente de ponderação. Esta heurística visa trocar os “piores” elementos da solução com os “melhores” elementos fora da solução (vizinhança).

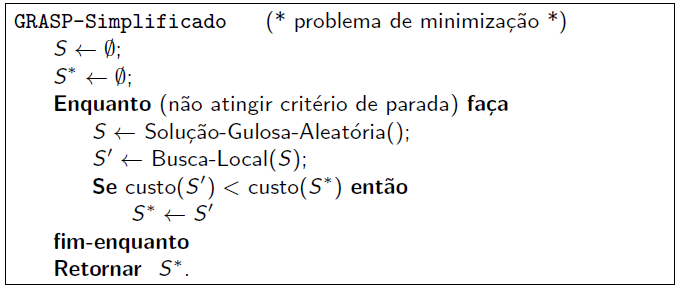
* **Busca local:**

Colocamos um limite de tempo (*LOCAL\_TIME\_LIMIT*) para a busca local visando diminuir seu tempo de execução para gerarmos mais soluções aleatórias, pois verificamos que muito dificilmente pararia por atingir um mínimo local.

1. **Descrição da implementação:**

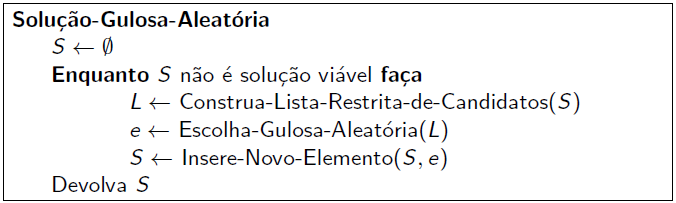
Para explicarmos melhor as heurísticas explicarei utilizando os slides das aulas e referências a nossa implementação no formato (Classe. método), indicando onde cada heurística se encaixa, seus motivos e estrutura de dados. Achamos isso suficiente para o entendimento da nossa implementação, para mais detalhes basta olhar o código que está todo comentado.

Para o GRASP (Grasp.execute):



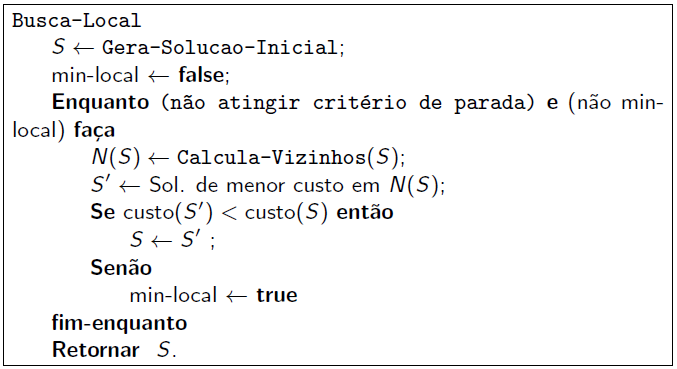
A nossa implementação está igual ao algoritmo do slide, a única coisa indefinida é critério de parada para o GRASP que utilizamos apenas o tempo, ou seja, enquanto não tiver passado 55 segundos após a leitura do arquivo de entrada (deixamos uma folga de 5 segundos).

Para a solução gulosa aleatória (Grasp.randomGreedySolution) :



Nosso implementação apresenta apenas uma diferença, montamos a lista de restrita de candidatos (Grasp.buildRestrictCanditatesList) apenas uma vez fora do loop, visando otimizar o tempo de execução do algoritmo, esta lista restrita de candidatos possui todos as torres possíveis ordenadas crescentemente por ponderação. Nossa escolha gulosa aleatória (Grasp.randomGreegyElement) escolhe aleatoriamente apenas um dos primeiros elementos da lista restrita, respeitando o limite da constante *RANDOM\_LIMIT,* removendo-o da lista para ele não ser escolhido novamente. E finalmente incluíamos o elemento escolhido na nossa solução (StationList) .

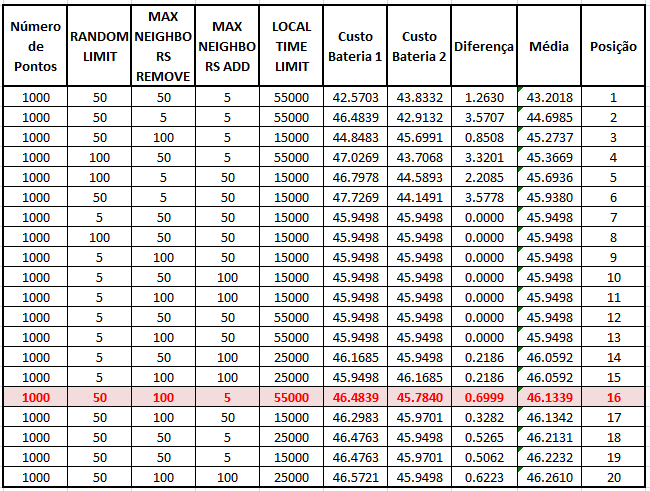
Para a busca local (Grasp.localSearch):

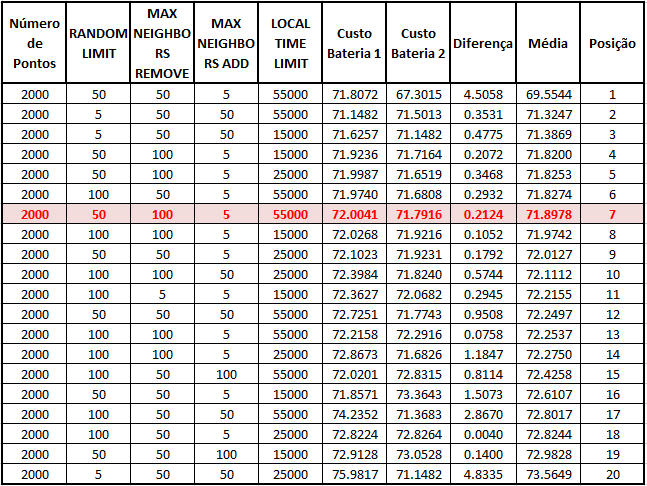


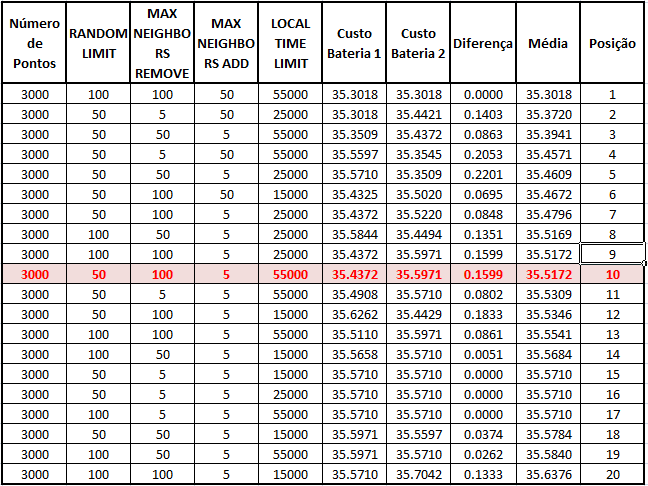
Nosso implementação apresenta apenas uma diferença, nós calculamos os vizinhos e procuramos a solução de menor custo ao mesmo tempo (Grasp.getBestNeighbor). Nossa solução inicial é a solução gulosa aleatória explicada anteriormente, e nosso critério de parada para a busca local, além o de alcançar o mínimo local, é o tempo, enquanto não tiver passado 55 segundos desde a leitura da entrada ou ter passado *LOCAL\_TIME\_LIMIT* desde a chamada da busca local. Nosso cálculo de vizinhos (Grasp.getBestNeighbor) troca os primeiros *MAX\_NEIGHBORS\_REMOVE* da solução, ordenados decrescentemente por ponderação, com os primeiros *MAX\_NEIGHBORS\_ADD* que não pertence à solução*,* ordenados crescentemente por ponderação. Lembrando que trocamos sempre um elemento por vez, também testamos uma nova solução sem aquele elemento da solução atual e ao mesmo tempo que trocamos procuramos pela melhor solução.

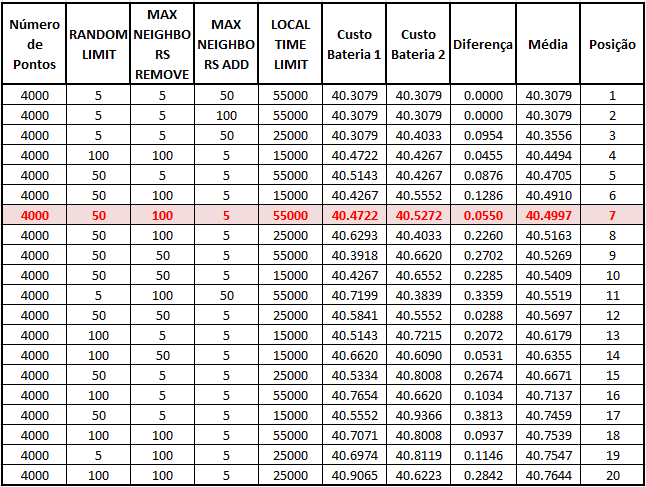
1. **Resultados computacionais:**

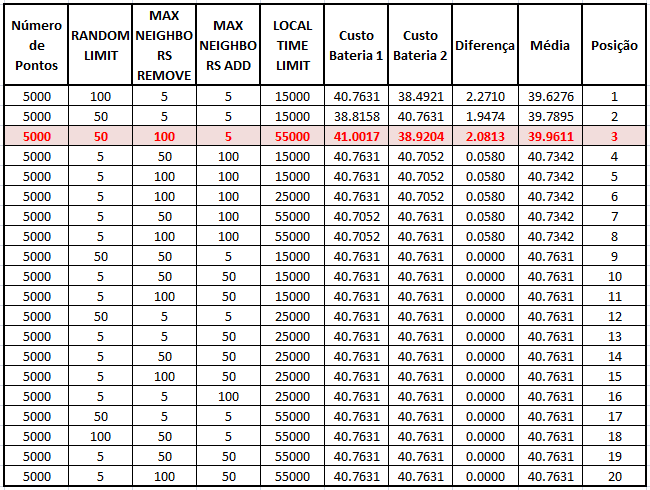
Fizemos duas baterias de testes do nosso programa para cada arquivo de entrada, onde cada teste possui diferentes configurações das constantes utilizadas no nosso programa. Cada arquivo possui 81 testes e cada teste durava aproximadamente 1 minuto, logo cada para rodar cada arquivo durava 1 hora e 20 minutos, como tínhamos cinco arquivos de entrada (com tamanhos 1000, 2000, 3000, 4000 e 5000) cada bateria durava 6 horas e 45 minutos. Com as duas baterias de testes fizemos uma média de cada configuração e classificamo-las de acordo com o valor do custo da solução, obtendo as tabelas abaixo (exibindo os 20 primeiros resultados de cada arquivo, pois os todos os 81 resultados não caberiam em 10 páginas):











A linha em vermelho de cada tabela corresponde à melhor configuração, escolhida para rodarmos nosso programa. Esta configuração foi a melhor dentre as 20 primeiras configurações em todos os arquivos, foram encontradas apenas 2 configurações dentre as 20 primeiras. Resolvemos escolher a configuração desta maneira porque percebemos que não existem diferenças muito grandes entre as 20 primeiras configurações de cada arquivo, a diferença fica em torno de 1% a 3.5% da melhor configuração.