UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES

RELATÓRIO DA PARTE II DO TRABALHO APRESENTADO À DISCIPLINA

ACH2016 – Inteligência Artificial

Integrantes:	NºUSP
André Ramos	9125339
Matheus Mendes de Sant'Ana	8921666
Victor Luiz Soares Alexandre Pereira	8921350

SÃO PAULO

2016

1-) Resumo:

Este relatório visa descrever e detalhar todo o processo da elaboração de um algoritmo para solucionar o Problema de Roteamento de Veículos (Vehicle Routing Problem – VRP), que será explicado mais adiante. Para isso, será implementado em Java o método meta-heurístico Simulated Annealing, e ainda serão usados três métodos para **geração de vizinhos**: One to One Exchange, Delete & Insert e Partial Reversal que serão descritos posteriormente, apresentando, para todos os códigos, seus respectivos pseudocódigos e uma descrição de como funcionam os algoritmos em si. Para fins comparativos, será implementado também o algoritmo heurístico **Centroid-Based**. Após isso, serão mostrados os diagramas de classes dos algoritmos, e as dificuldades encontradas durante toda a implementação das soluções. Serão expostas as configurações utilizadas nos experimentos, e ainda, uma tabela com os resultados obtidos. Por fim, faremos uma análise geral dos resultados com auxílio de gráficos e uma breve conclusão de todo o trabalho desenvolvido.

2-) Introdução:

O problema de roteamento de veículos (VRP – Vehicle Routing Problem) é uma otimização de um problema de programação combinatória e de programação com números inteiros. O algoritmo busca atender um número de clientes com uma frota limitada de veículos, com base em um ou vários pontos de entrega que devem ser determinados por uma série de cidades ou clientes geograficamente dispersos, com o objetivo de entregar produtos a todo esse conjunto de clientes, com demandas conhecidas, em rotas de veículos de custo mínimo com origem e destino em um depósito. Proposto por George Dantzig e John Ramser em 1959, VRP é um problema importante nas áreas de transporte, distribuição, logística e socioambientalismo, pois um roteamento bem escolhido implica em menos emissão de poluentes, maior eficiência na execução do trabalho e menor gasto para a empresa. **OBS:** Antes de executar os programas Java, certifique-se de alterar no método "main" de cada programa o diretório onde está(ão) a(s) entrada(s).

3-) Descrição do algoritmo Centroid VRP:

1ª etapa - Criar clusters: Encontrar vértice mais distante de "depot" (ideia de que esta "semente" pode ser o ponto crítico onde podem ser satisfeitas as demandas de outros vértices distantes), após isso, preencher o cluster com vértices mais próximos do centro geométrico deste cluster que não excedam a capacidade do caminhão e que não estejam em outro cluster. A cada inserção de um vértice em um cluster, atualizar o centro geométrico que terá como valor X a média dos valores X de cada vértice do cluster; e o valor Y como a média dos valores Y de cada vértice do cluster. Caso o vértice mais próximo do centro geométrico tenha demanda acima da demanda atual disponível no caminhão, o algoritmo parte para a criação do próximo cluster, que consiste na mesma implementação de encontrar o vértice mais distante de "depot" que não esteja em um cluster e preencher o cluster seguindo os critérios anteriores.

2ª etapa – Reorganizar clusters: Encontrar vértices que possuam maior proximidade com outros clusters do que com o cluster no qual está inserido. Se este vértice não for exceder a capacidade do caminhão deste outro cluster que possui centro geométrico mais próximo, retirá-lo do cluster anterior e inserí-lo neste melhor cluster encontrado. Atualizar centro geométrico dos clusters que foram modificados (X = média de X dos vértices do cluster e Y = média de Y dos vértices do cluster).

3ª etapa – Definir uma rota entre "depot" e cada um dos clusters, sendo que o caminhão deve partir de "depot", percorrer o cluster e voltar ao "depot". A ordem entre os vértices do cluster (exceto o primeiro e o último vértice serem "depot") deve ser definida através de um algoritmo que implemente o Problema do Caxeiro-Viajante ou TSP.

Algoritmo final: Em um laço de repetição (de repetições definidas pelo usuário), aplicar as etapas 1 e 3; para cada fim da repetição do laço deve-se aplicar a etapa 2 (reorganização dos clusters) e retornar como resultado a melhor solução dentre todas as repetições do laço.

4-) Pseudocódigos e descrição dos algoritmos implementados:

A seguir serão apresentados descrições e seus respectivos pseudocódigos de cada algoritmo implementado para a solução do CVRP. As descrições e o pseudocódigo referentes ao Simulated Annealing já foram citados na entrega da primeira parte do EP. Sendo assim, para evitarmos repetições, focaremos em detalhar os algoritmos inéditos. A seguir são apresentados um algoritmo do Simulated Annealing otimizado e os três métodos de geração de vizinhos:

4.1-) Novo Simulated Annealing (v2):

```
Para cada vértice i do grafo faça { //Repassa todos os estados para a lista atual
        se vértice "i" não é depot {
                sequenciaEstados.add(i); //Adiciona todos os estados menos o inicial
        }
}
sequenciaEstadosAntiga ← copiar(sequenciaEstados);

tamanhoLista = tamanho(sequenciaEstados);
temperatura = tamanhoLista;
taxaResfriamento =  1/(tamanhoLista*tamanhoLista);
limiteTemperatura = taxaResfriamento;
ultimaModificacao = (tamanhoLista*tamanhoLista);
```

```
distTotalAnterior;
distTotalAtual;
caminhoesUsadosAnterior = 0;
caminhoesUsadosAtual = 0;
solucaoAtual ← transformarEmSolucao(sequenciaEstados); //Atualiza solucao
distTotalAnterior = custoTotal(solucaoAtual);
caminhoesUsadosAnterior = tamanho(solucaoAtual);

Enquanto temperatura >= limiteTemperatura OU ultimaModificacao > 0 faça:
        gerarVizinhosAleatoriamente(sequenciaEstados);
        solucaoAnterior ← transformarEmSolucao(solucaoAtual);
        solucaoAtual ← transformarEmSolucao(sequenciaEstados);
        distTotalAtual = custoTotal(solucaoAtual);
        caminhoesUsadosAtual = tamanho(solucaoAtual);
        modificou = false;
        Se (distTotalAtual > distTotalAnterior)  { //Solucao atual eh pior que a anterior
                aceitacao = aceita(distTotalAnterior, distTotalAtual, temperatura);
                Se aceitacao==false { //Retoma a condicao inicial
                        sequenciaEstados ← copiar(sequenciaEstadosAntiga);
                        solucaoAtual ← transformarEmSolucao(solucaoAnterior);
                        modificou = false;
                }
                senão {
                        sequenciaEstadosAntiga ← copiar(sequenciaEstados);
                        distTotalAnterior ← distTotalAtual;
                        caminhoesUsadosAnterior ← caminhoesUsadosAtual;
                        modificou = true;
                }
        }
        senão se distTotalAtual == distTotalAnterior
                //Se melhorou ou se nao fez diferenca, a solucao sera a atual
                Se caminhoesUsadosAtual <= caminhoesUsadosAnterior {
                        sequenciaEstadosAntiga ← copiar(sequenciaEstados);
```

```
                    distTotalAnterior ← distTotalAtual;
                    caminhoesUsadosAnterior ← caminhoesUsadosAtual;
                    modificou = true;
                }
                senão {
                    sequenciaEstados ← copiar(sequenciaEstadosAntiga);
                    solucaoAtual ← solucaoAnterior;
                    modificou = false;
                }
            }
            senão { //Solucao atual eh melhor que a anterior
                sequenciaEstadosAntiga ← copiar(sequenciaEstados);
                distTotalAnterior ← distTotalAtual;
                caminhoesUsadosAnterior ← caminhoesUsadosAtual;
                modificou = true;
            }
            Se modificou == true {
                 ultimaModificacao ++;
            }
            senão ultimaModificacao --;
            temperatura = temperatura - (temperatura*taxaResfriamento);
        fim enquanto
    fim
```

4.1.1-) One to One Exchange:

```
    oneToOneExchange (posição1, posição2, Lista de estados):
        //Executa a troca de estados
        swap1 ←  estados.get(posição1);
        swap2 ← estados.get(posição2);
        estados.set(posição1, swap2);
        estados.set(posição2, swap1);
    fim
```

4.1.2-) Delete & Insert:

```
deleteAndInsert (posDelete, posInsert,  Lista de estados)
        valorEstado ← estados.get(posDelete);

        estados.remove(posDelete);

        estados.add(posInsert, valorEstado);

fim
```

4.1.3-) Partial Reversal:

```
partialReversal (int pos1, int pos2,  Lista de estados)
        posEsquerda;

        posDireita;

        swap1;

        swap2;

        Se pos1 < pos2 {

                posEsquerda ← pos1;

                posDireita ← pos2;

        }

        senão { //pos2 < pos1

                posEsquerda ← pos2;

                posDireita ← pos1;

        }

        Enquanto posEsquerda < posDireita {

                swap1 = estados.get(posEsquerda);

                swap2 = estados.get(posDireita);

                estados.set(posEsquerda, swap2);

                estados.set(posDireita, swap1);

                posEsquerda++;

                posDireita--;

        }

fim
```

4.2-) Centroid-Based:

Clusters (Centroide):

```
        Conjunto de Clusters configuracaoAtual = construirClusters();
        Conjunto de Clusters melhorConfiguracao;

        estabelecerRota(configuracaoAtual);
        menorDistancia = custoTotalCluster(configuracaoAtual);
        distanciaAtual = menorDistancia;
        removerDepots(configuracaoAtual);

    melhorConfiguracao = copiar(configuracaoAtual);
        modificou = false;
        Enquanto true faça:
            modificou = ajustarClusters(configuracaoAtual);
            Se !modificou break;
            estabelecerRota(configuracaoAtual);
            distanciaAtual = custoTotalClusters(configuracaoAtual);
            Se distanciaAtual < menorDistancia {
                menorDistancia = distanciaAtual;
                removerDepots(configuracaoAtual);
                melhorConfiguracao = copiar(configuracaoAtual);
            }

            senão {
                removerDepots(configuracaoAtual);
            }
        fim laço
        Para cada cluster (clusterAtual) de melhorConfiguracao {
            clusterAtual.addFirst(indexDepot);
            clusterAtual.addLast(indexDepot);
        }
        copiarDeListaParaSolucao(melhorConfiguracao);
    fim
```

5-) Implementação:

Para uma melhor visibilidade, os diagramas dos algoritmos implementados foram postos em arquivos **pdf** separados e estão localizados junto à pasta deste relatório: os elementos precedidos por um círculo verde são públicos enquanto os elementos precedidos por um quadrado vermelho são privados. Vale lembrar que os diagramas não tiveram muitas ligações entre as classes pois a maioria dos métodos e atributos são do tipo "static" nos 3 projetos (EP1 antigo, EP2 com método para gerar vizinhos e EP2 centroide).

5.1-) Dificuldades encontradas durante a implementação:

1-) No **Simulated Annealing v2**: encontrar uma maneira de utilizar aleatoriamente um dos 3 métodos para gerar vizinhos com probabilidade 1/3 cada um ("One to one exchange", "Partial Reversal" e "Delete and insert"). O método **random()** só foi utilizado para encontrar 2 vértices distintos, pois sua aleatoriedade é pouco previsível, o que nesta implementação não seria favorável à probabilidade praticamente uniforme (uniformidade essa proposta no artigo de *H. Kokubugata* e *H. Kawashima*) de chamar um dos 3 métodos. A solução foi utilizar o método **nextGaussian()** que gera um número de acordo com a probabilidade da distribuição normal; através desse método modelamos 3 intervalos de confiança para obtermos a probabilidade uniforme de 1/3 para cada um dos métodos: valores **menores que -0,431**; valores **entre -0,431 e 0,431**; e valores **maiores que 0,431**. A variável responsável por essa comparação de probabilidade uniforme se chama "valorAleatorio", presente no método "gerarVizinhosAleatoriamente" da classe "GerarVizinhos".

2-) A hipótese de que o algoritmo **Simulated Annealing v2** estava falhando devido aos resultados possuírem um custo muito melhor em relação ao algoritmo **v1**. Constatamos então que o método "Partial Reversal" é que tornava os resultados tão discrepantes em relação ao algoritmo anterior, pois de uma determinada iteração para a outra, uma hipótese de solução pode mudar quase por completo, dependendo do "pedaço" da rota que foi retirado e recolocado inversamente na solução corrente, melhorando o algoritmo.

3-) A última dificuldade foi encontrar um bom algoritmo que resolvesse o TSP (problema do caxeiro viajante) para aplicar a terceira etapa do algoritmo Centroid-Based, que consiste em estabelecer uma rota com uma determinada ordem dentro de cada cluster, tendo os nós inicial e final iguais a depot". A solução encontrada foi utilizar os algoritmos **Brute Force** (força bruta) de complexidade de tempo **O(n!)** que retorna a solução ótima de uma rota para uma sequência de estados; e o algoritmo **Simulated Annealing** de complexidade de tempo em torno de $\Omega(n^2)$ e $O(n^3)$ (vide na **seção 8** "Extra" o cálculo da complexidade) deduzida pela temperatura = **n** e pela taxa de resfriamento = $1/n^2$ em que **n** é o número de estados menos depot (não é possível saber a complexidade precisa, pois o número de iterações do algoritmo é aleatório: depende também da variável que conta o **número de modificações**). Analisamos o tempo gasto para o algoritmo Brute Force, e inferimos que o melhor seria utilizar esse algoritmo para clusters que possuam quantidade limite de vértices igual a **8** (**sem contar os vértices "depot"** do início e do fim), o que nos traz um gasto aceitável de tempo para o Centroid-Based, de no máximo 1 segundo para todas as entradas testadas

(utilizando o algoritmo Brute Force para clusters com 9 vértices, o Centroid-Based passa a gastar em média 10 segundos ou mais para cada arquivo de entrada executado). Logo, para um cluster que possui quantidade de vértices menor ou igual a 8 (sem contar os "depots"), é gerada uma rota utilizando o algoritmo **Brute Force**; do contrário, a rota será gerada utilizando o algoritmo **Simulated Annealing**.

6-) Experimentos – Configuração:

Em seguida serão realizados testes com todos os algoritmos implementados para fins comparativos. Sendo assim, para atingir resultados condizentes, serão usados os mesmos parâmetros e a mesma máquina para todos os testes.

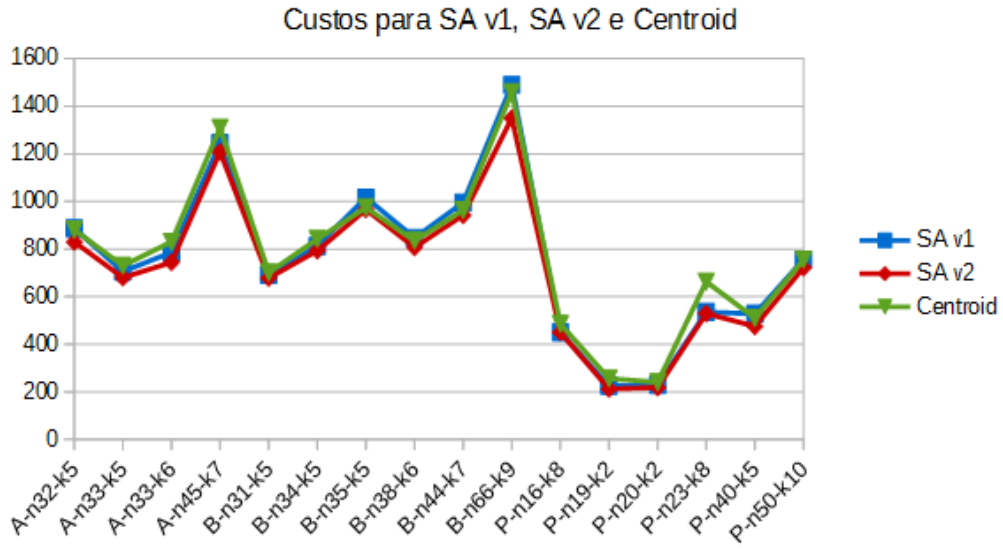
A configuração da máquina utilizada para a realização dos testes foi: Notebook Sony VAIO Intel(R) Core(TM) i5 CPU M 520 @ 2.40GHz, com 8 GB de memória RAM e 500GB de HD. O SO utilizado foi Windows 7 Ultimate 64bits.

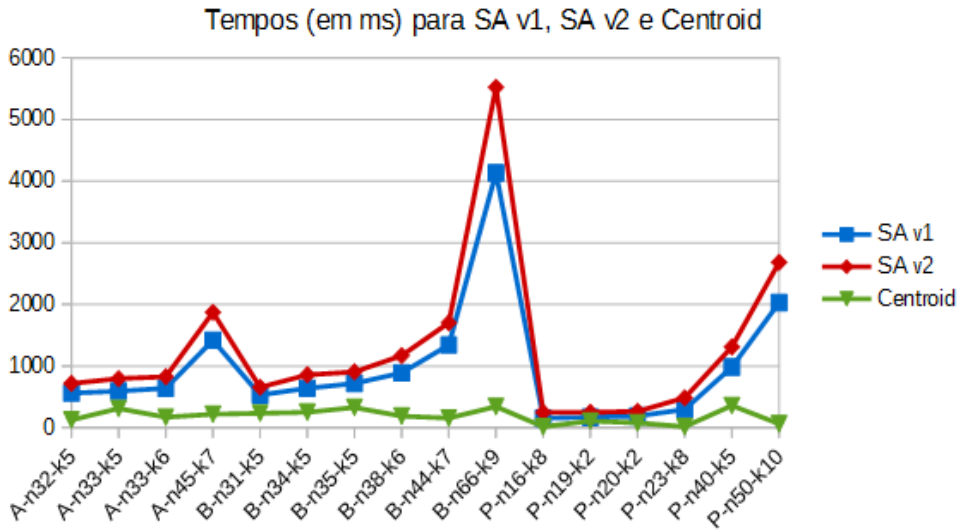
Foi escolhido o valor de 10 iterações nos algoritmos para a busca do melhor resultado. Caso o resultado não satisfaça os critérios exigidos, pode-se aumentar o número de iterações, tendo em mente que quanto maior o número de iterações, mais tempo levará para que o algoritmo retorne um resultado.

6.1-) Experimentos – Resultados e Soluções Ótimas:

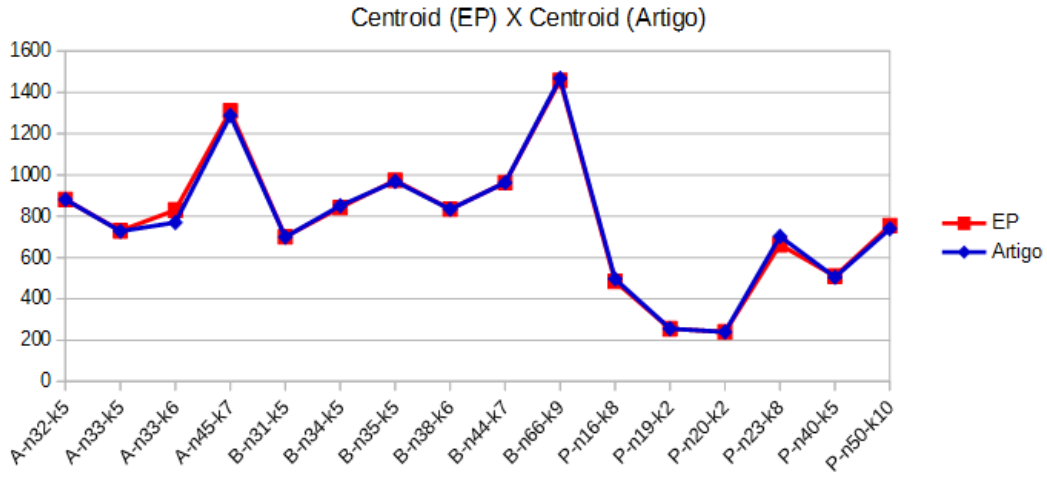
	SA Antigo		SA Gerar Vizinhos		Centroid		Solução Ótima ou Melhor Encontrada
	Custo	Tempo	Custo	Tempo	Custo	Tempo	Custo
A-n32-k5	**887**	562	**830**	717	**881**	125	**784**
A-n33-k5	**705**	593	**680**	796	**730**	312	**661**
A-n33-k6	**786**	640	**744**	827	**830**	171	**742**
A-n45-k7	**1245**	1419	**1208**	1872	**1310**	218	**1146**
B-n31-k5	**691**	532	**678**	655	**701**	234	**672**
B-n34-k5	**813**	640	**793**	858	**842**	250	**788**
B-n35-k5	**1016**	718	**966**	905	**974**	328	**955**
B-n38-k6	**847**	889	**807**	1170	**835**	187	**805**
B-n44-k7	**995**	1342	**942**	1703	**963**	156	**909**
B-n66-k9	**1490**	4134	**1350**	5523	**1458**	343	**1374**
P-n16-k8	**450**	156	**450**	250	**485**	16	**435**
P-n19-k2	**225**	171	**212**	249	**256**	109	**212**
P-n20-k2	**229**	188	**218**	265	**240**	78	**220**
P-n23-k8	**535**	296	**529**	484	**663**	16	**554**
P-n40-k5	**528**	983	**475**	1311	**510**	359	**458**
P-n50-k10	**756**	2028	**722**	2683	**755**	62	**696**

6.2-) Análise dos resultados obtidos:





Entradas	Centroid (EP)	Centroid (Artigo)
A-n32-k5	881	881
A-n33-k5	730	728
A-n33-k6	830	770
A-n45-k7	1310	1288
B-n31-k5	701	700
B-n34-k5	842	851
B-n35-k5	974	969
B-n38-k6	835	834
B-n44-k7	963	963
B-n66-k9	1458	1468
P-n16-k8	485	497
P-n19-k2	256	256
P-n20-k2	240	240
P-n23-k8	663	703
P-n40-k5	510	505
P-n50-k10	755	740



Nos dois primeiros gráficos, ficou evidente que o **Simulated Annealing v2** apresentou os **melhores resultados** para o **custo** para todas as entradas mas não para o **tempo gasto**. Os valores para o **Simulated Annealing v1** e **Centroid-Based** foram similares para todas as entradas; porém para as entradas **A-n32-k5**, **B-n35-k5**, **B-n38-k6**, **B-n44-k7**, **B-n66-k9**, **P-n40-k5 e P-n50-k10**, os resultados foram melhores para o algoritmo **Centroid-Based** do que para o algoritmo **Simulated Annealing v1**. Para as outras instâncias além dessas citadas, o algoritmo **Simulated Annealing v1** se destacou em relação ao Centroid; ressaltamos por último que em termos de tempo o algoritmo **Centroid-Based** foi o melhor em todas as entradas do que os outros dois algoritmos.

É possível notar nos gráficos que os resultados obtidos no tempo gasto da execução do algoritmo **Centroid-Based** se mantiveram quase constantes, enquanto as duas versões do **Simulated Annealing** apresentaram comportamento muito parecido, com a **versão 2** tendo um rendimento um pouco superior. A comparação do custo entre o algoritmo Centroid-Based desenvolvido por nós e o artigo fornecido evidencia um desempenho praticamente idêntico.

7-) Conclusões:

Através dos fatos analisados, percebemos que depois de acompanhar o desenvolvimento conceitual e prático dos algoritmos abordados, com auxílio dos testes realizados, o algoritmo **Centroid-Based** se destacou principalmente pela sua eficiência (custo/tempo), obtendo os melhores resultados no tempo gasto e também superando resultados para várias entradas se comparado ao algoritmos **Simulated Annealing v1**.

O algoritmo **Simulated Annealing v2**, desenvolvido na parte II deste EP, obteve resultados superiores no aspecto custo do resultado **em relação à versão 1 e em relação ao algoritmo Centroid**: como mencionado na *seção 5.3*, a melhoria da versão 2 desse algoritmo se dá intuitivamente pelo método "Partial Reversal", o qual pode alterar drasticamente as hipóteses sobre soluções melhores entre duas soluções geradas de um momento para o outro, o contrário da **versão 1 que é mais estável nesse aspecto** (pois só utiliza "swap" entre 2 nós, que equivaleria ao método "one-to-one exchange"). A versão 2 do Simulated Annealing apresentou resultados de custo excelentes: para várias entradas, após testarmos entre 1 e 10 vezes

este algoritmo, obtivemos resultados ótimos e até mesmo resultados melhores que alguns já descobertos (quando não são ótimos).

Em suma, para uma aplicação real que necessite melhores resultados a qualquer custo, propomos um algoritmo que execute os 3 algoritmos citados neste artigo, recebendo como resultado final **o melhor resultado dentre os resultados desses três algoritmos**. Mesmo que o algoritmo Simulated Annealing v2 tenha se saído melhor para todos os testes com relação ao resultado, pode ser que em algum momento algum dos outros dois algoritmos (Simulated Annealing v1 ou Centroid) se saiam melhor. Já para aplicações cujo **tempo gasto** seja o fator mais relevante, o algoritmo **Centroid-Based** é sem dúvida a melhor opção, pois une o menor gasto de tempo a uma solução de baixo custo.

8-) Extra:

8.1-) Quantidade de veículos acima do ótimo esperado:

Em grande parte dos casos de teste, podem ocorrer situações em que o número de veículos utilizados está acima do ótimo definido, principalmente pela maneira como o algoritmo Centroid-Based foi definido no artigo (não há, por exemplo, casos em que o algoritmo volta atrás para verificar se há uma solução que utiliza menos veículos). Podemos dizer também que para o algoritmo Centroid-Based não há a **fusão de clusters**, o que seria uma interessante aplicação para otimizar o algoritmo aqui proposto, pois para obtermos menor custo poderíamos, por exemplo, fundir dois ou mais clusters se a demanda total somada deles não ultrapassasse a demanda limite de um veículo: essa proposta fica a critério de novas implementações.

8.2-) Complexidade aproximada do algoritmo Simulated Annealing:

Os principais fatores que atuam na complexidade de tempo do Simulated Annealing são as variáveis “temperatura”, “taxaResfriamento” e “limiteTemperatura” (além da variável “ultimaModificacao” que depende de aleatoriedade, logo não será mencionada nessa seção para maior simplicidade). Para tal, consideraremos 5 iterações do algoritmo considerando apenas o código abaixo:

```
public class SimplificacaoSA {
    public static void main(String[] args) {
        double temp = 100;
        double taxa = 1/(temp*temp);
        double limite = taxa;

        int cont=0;
        while(temp >= limite) {
            temp=temp-(temp*taxa);
            cont++;
        }
        System.out.println(cont);
    }
}
```

temp = temperatura; **taxa** = taxa de resfriamento; **limite** = limite que a temperatura pode atingir.

Iterações:

1ª temp1 = **1*temp – 1*temp*taxa**

2ª temp2 = (temp – temp*taxa) – (temp – temp*taxa)*taxa =

= **1*temp – 2*temp*taxa + 1*temp*taxa²**

3ª temp3 = **1*temp – 3*temp*taxa + 3*temp*taxa² – 1*temp*taxa³**

4ª temp4 = **1*temp – 4*temp*taxa + 6*temp*taxa² – 4*temp*taxa³ + 1*temp*taxa^4**

5ª temp5 = **1*temp – 5*temp*taxa + 10temp*taxa² – 10temp*taxa³ + 5*temp*taxa^4 – 1*temp*taxa^5**

Percebemos nestas iterações 3 fenômenos: cada iteração possui **n+1 termos**; os **termos ímpares são positivos e os pares são negativos** e as constantes que multiplicam cada termo obedecem à regra do **triângulo de Pascal**, sublinhadas para maior clareza. Para calcularmos a complexidade de tempo a partir de uma **temperatura de valor "n"**, temos a seguinte inequação que nos afirma que o número de iterações findará quando a temperatura for maior ou igual ao limite de temperatura:

Última iteração:

-Lembre-se que "temp" é a temperatura inicial

-Um termo **(x z)** significa o termo binomial **x!/z!(x-z)!**

limite <= temp + $(-1)^1 * (i\ 1)*temp*taxa + (-1)^2 * (i\ 2)*temp*taxa^2 + (-1)^3 * (i\ 3)*temp*taxa^3 + \dots$

$+ (-1)^i * (i\ i)*temp*taxa^i$

Dado **n** o número de vértices do problema VRP, substituindo **limite** por **1/n²**; **taxa** por **1/n²**; **temp** por **n** e calculando operações possíveis temos:

$1/n^2 <= n - (i\ 1)*n*1/n^2 + (i\ 2)*n*(1/n^2)^2 - (i\ 3)*n*(1/n^2)^3 + \dots + (-1)^i *n*(1/n^2)^i$

$1/n^2 <= n + \Sigma\ (-1)^k *(i\ k)*n*(1/n^2)^k$ → somatório indo de **k=1** até **i**

$1/n^2 - n <= \Sigma\ (-1)^k *(i\ k)*n*(1/n^2)^k$

$1/n^2 - n <= n*\Sigma\ (-1)^k *(i\ k)*(1/n^2)^k$

$1/n^3 - 1 <= \Sigma\ (-1)^k *(i\ k)*(1/n^2)^k$

Suposição de uma aproximação: definir somatório como uma **PG infinita**, na qual sabemos que metade dos termos são positivos e a outra metade é negativa (como demonstrado acima), logo a razão será

negativa e o primeiro termo será negativo. Dado que o **primeiro termo é negativo**, o **segundo termo positivo seria aproximadamente o primeiro termo vezes $-1/n^2$** e daí por diante. Para essa PG, assumiremos que **$q = -1/n^2$** (razão) e também assumiremos que **$a1 = -i/n^2$** (1º termo da somatória da inequação anterior):

Soma de infinitos termos de uma PG infinita → $S = a1/(1-q)$

$S = (-i/n^2) / [1-(-1/n^2)]$

$S = (-i/n^2) / [(n^2+1)/n^2]$

$S = -i/(n^2+1)$

Fazendo a inequação deste somatório obtido com o termo da esquerda da inequação da complexidade temos:

$1/n^3 - 1 <= -i(n^2+1)$

$-i >= (n^2+1)/n^3 -(n^2+1)$

$-i >= (n^2+1)/n^3 - n^2 - 1$

Aproximando $(n^2+1)/n^3$ para 0 temos:

$-i >= -n^2 -1$

$i >= n^2 + 1$

Temos então que a complexidade de tempo tem limite inferior em torno de **$\Omega(n^2)$**

Através dos modelos matemáticos apresentados, notamos como funciona a complexidade do algoritmo que tem seus termos crescendo conforme ocorre o crescimento de iterações. Utilizando um programa, poder-se-ia estimar valores para "i" e para o somatório do **lado direito da inequação** até que se chegue em um resultado próximo ao **lado esquerdo da inequação**, obtendo um resultado mais preciso da complexidade de tempo do que a hipótese aqui demonstrada.

Como a inequação não tem solução trivial, para fins práticos, podemos estimar a complexidade através de 3 temperaturas diferentes executando o código mencionado anteriormente. Para as temperaturas 10, 100 e 1000 temos 688, 138149 e 20723256 iterações respectivamente, com logaritmos em suas respectivas bases aproximadamente iguais a 2.83, 2.57 e 2.43 de onde tiramos que a complexidade de tempo aproximada do Simulated Annealing implementado está entre **$\Omega(n^2)$** – dado que nenhum desses 3 testes ficou abaixo do limite calculado matematicamente – e **$O(n^3)$**.

9-) Bibliografia:

https://www.prp.rei.unicamp.br/pibic/congressos/xiicongresso/cdrom/pdfN/386.pdf

http://www.sbmac.org.br/cnmacs/2004/cd_cnmac/files_pdf/10386a.pdf

http://www.seer.unirio.br/index.php/isys/article/view/5163/4917

http://www.bernabe.dorronsoro.es/vrp/

http://www.icmc.sc.usp.br/~sandra/G9_t2/annealing.htm

http://www.lac.inpe.br/~lorena/cap/Aula_C01.pdf

https://en.wikipedia.org/wiki/Simulated_annealing

http://cdn.intechweb.org/pdfs/4627.pdf

Han, S. A Centroid-Based heuristic algorithm for the capacitated vehicle routing problem. Computing and Informatics, Vol. 30, 2011, p. 721-732.

H. Kokubugata and H. Kawashima. Application of Simulated Annealing to Routing Problems in City Logistics. In Simulated Annealing, Book edited by: Cher Ming Tan, ISBN 978-953- 7619-07-7, pp. 420, February 2008, I-Tech Education and Publishing, Vienna, Austria