Instituto de Computação

Projeto e Análise de Algoritmos II

Trabalho Prático: Relatório

Prof. Eduardo C. Xavier

Alunos:

Alline Kobayashi RA: 070058

Luiz Lemos RA: 044916

Virgílio Santos RA: 036470

Índice

1. Introdução
2. Código Fonte

2.1**. Estrutura de Classes**

2.2**. Principais Métodos**
3. Heurística Utilizada

3.1**. Simulated Annealing**

3.2**. Descrição de Alto Nível**

3.3**. Adaptações utilizadas**
4. Resultados

4.1**. Testes Realizados**

4.2**. Soluções Encontradas e Tempo de Execução**
5. Conclusão

1. Introdução

O estudo de complexidade de algoritmos nos mostra que existem problemas que são particurlarmente difíceis de se encontrar uma solução. Isto pode se dever a uma dificuldade intrínseca do problema ou ainda ao desconhecimento de algoritmos que resolvam bem estes problemas, neste segundo caso temos a classe NP.

Temos para essa classe, no entanto, métodos para aproximar a solução, algoritmos aproximativos, ou uma solução possivelmente boa em tempo útil, heurística. Tentaremos resolver o problema de minimização Set-Covering com custos utilizando da heurística Simulated Annealing.

2. Código Fonte

2.1. **Estrutura de Classes**

Como foi utilizado Java, fizemos o programa completamente orientado a objetos, o javadoc mais atual pode ser encontrado em:

http://www.students.ic.unicamp.br/~ra036470/javadoc/

A estrutura de classes utilizada foi a seguinte:

Spot: Cada ponto dado tem como parâmetros:

- **spotNumber:** inteiro identificador do ponto;
- **stations:** ArrayList<Station> de estações que cobrem tal ponto

Station: Cada estação dada e tem como parâmetros:

- **stationId:** String que identifica a estação;
- **stationCost:** double que representa o custo da estação;
- **coveredSpots:** ArrayList<Spot> conjunto de pontos cobertos por esta estação.

Instance: Representação da instância dada ao iniciar o problema. Parâmetros:

- **numberOfSpots:** inteiro que representa o número total de pontos dados;
- **numberOfStations:** inteiro que representa o número total de estações;
- **stations:** ArrayList<Station> conjunto de estações do problema;
- **spots:** ArrayList<Spot> conjunto de pontos da instância.

Solution: Um conjunto de estações que forma uma solução qualquer.
Parâmetros:

- **custo**: double com a soma dos custos de todas as estações desta solução;

- **stationSet:** ArrayList<Station> com o conjunto de estações da solução;
- **spotsCovering:** ArrayList<Integer> Trata-se de uma lista com o número de estações, dentres as estações pertencentes à solução em questão, que cobrem cada ponto. O índice da lista é o identificador do ponto e o valor é o número de estações que cobrem tal ponto.

Annealing: Essa classe contém os métodos principais para o funcionamento do algoritmo. Detalhes sobre sua implementação podem ser encontrados no link para sua documentação; são eles:

- **solucaoInicial**(Instance inst, Solution s)**:**

Gera solução inicial para ser utilizada na heurística;

- **gerarVizinhanca** (Instance inst, Solution curr, Solution neig) **:**

Algoritimo para gerar as vizinhaças. A distância entre a solução corrente e o próximo vizinho é dada por uma porcentagem.;

- **gerarVizinhanca2** (Instance inst, Solution curr, Solution neig)**:**

Esse foi o primeiro algoritmo que utilizamos para gerar vizinhos, ele basicamente deixa um dos pontos da solução descoberto e consequententemente alguns outros pontos e são então cobertos utilizando novas estações.

- **removerRedundancia** (Solution neig)**:**

Esse método privado (só utilizado internamente na classe) remove possíveis redundâncias na solução vizinha gerada.

- **aceitarSolucao** (double delta, double T, double k)**:**

Esse método simplesmente responde a probabilidade de aceitar ou não uma solução não necessariamente melhor que a corrente.

InstanceReader: Essa classe recebe em seu construtor o nome do arquivo de instância e monta toda a estrutura de dados. Foi feita dessa maneira e não como um método de Instance por questão de estilo de forma que há uma camada de montagem da Instância, a Instância em si e por fim os pacotes de algorítmos. Seus parâmetros são os mesmos da classe instância e :

- **instances**: BufferedReader que representa o arquivo sendo tratado;

Há outros métodos não citados nesse relatório já que são de menor importância ou são apenas ferramentais mas que podem ser vistos no javadoc disponibilizado ou mesmo no código fonte.

3. Heurística Utilizada

3.1. **Simulated Annealing**

Baseado em processo de criações de cristais o Simulated Annealing tenta se aproveitar de conceitos de física estatística para gerar boas soluções fugindo de mínimos locais.
Falando em termos práticos o Simulated Annealing começa a partir uma determinada solução inicial tida como ótima. A partir de então,substitui a solução atual por uma solução próxima, escolhida de acordo com uma função objetivo e com uma variável *T*, é o que chamamos de escolha do vizinho. Quanto maior for *T*, maior a componente aleatória que será incluída na próxima solução escolhida. À medida que o algoritmo progride, o valor de *T* é decrementado, começando o algoritmo a converter para uma solução ótima, necessariamente local.
Uma das principais vantagens deste algoritmo é permitir testar soluções mais distantes da solução atual e dar mais independência do ponto inicial da pesquisa. Selecionamos este método por ele ter uma iteração muito mais barata que métodos como Algoritmos Genéticos, uma capacidade de fuga de uma cela maior que o GRASP e mais clareza sobre o funcionamento dos processos aleatórios envolvidos.

3.2. **Descrição de Alto Nível**

A seguir é apresentado o pseudo-código do algoritmo utilizado por nós:

```
Simulated-Annealing
        L2 ← 0, L1 ← 0, n ← 0, times ← 0;
        T ← temp, k ← 1, alfa ← 0.99;
        initTime ← Tempo atual;
        currTime ← Tempo atual;

        curr ←solucaoInicial(inst, curr);
        best ← curr;

        Enquanto tempo de execução < 1 minuto
        .       Enquanto L1 < lim1
        .       .       Enquanto L2 < lim2
        .       .       .       neig ← gerarVizinhanca(inst, curr, neig);
        .       .       .       delta ← custo(neig) − custo(curr);
                .       .       .       Se delta < 0 então
        .       .       .       .       curr ← neig;
        .       .       .       .       currCustos ← currCustos U curr
        .       .       .       .       Se custo(curr) < custo(best) então
        .       .       .       .       best ← curr;
        .       .       .       .       bestCustos ← bestCustos U best
        .       .       .       Se não
        .       .       .       .       Se aceitarSolucao(delta, T, k) então
        .       .       .       .       .       curr ← neig;
        .       .       .       .       .       currCustos ← currCustos U curr;
        .       .       .       Incrementa L2;
        .       .       L2 ← 0;
        .       .       T ← alfa * T;
```

. . Incrementa L1;
. Incrementa n;
. Incrementa times;
. L1 ← 0;
. alfa ← alfa * 0.95;
. T ← temp * (10^n);
. Atualiza currTime;

3.3. **Adaptações utilizadas**

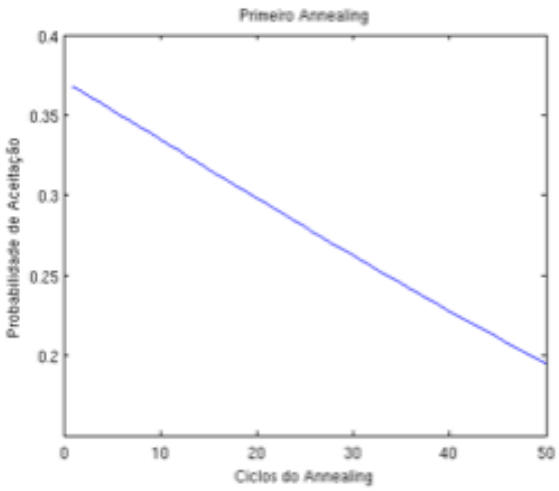
Foram feitas adaptações na heurística inicialmente escolhida afim que se aproveitasse bem o tempo limite dado. De forma simples podemos dizer que o annealing é rodado quantas vezes for possível no intervalo de 1 minuto com uma váriação de parâmetros sobre a aceitação de soluções aleatórias piores que a corrente.

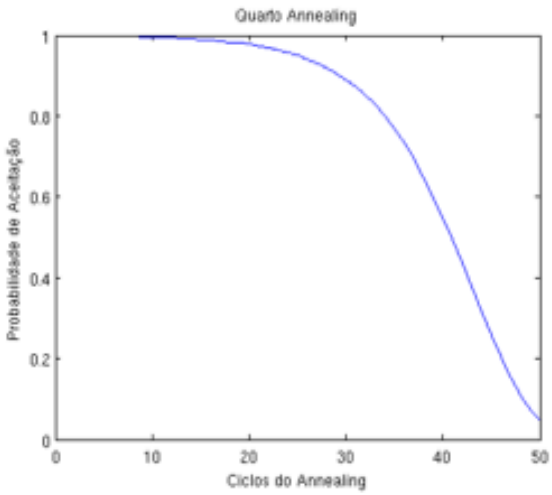
Afim de que este processo gerasse boas respostas aproveitamos do conceito de busca local e aleatoriedade para montar uma espécie de meta-annealing. O programa rodara inicialmente um annealing de comportamento bem pouco aleatório, ou seja, próximo de uma busca local. Gerando uma solução inicial para o annealing mais agressivo aleatóriamente seguinte e assim sucessivamente. Assim gerando respostas razoáveis para o tempo dado.

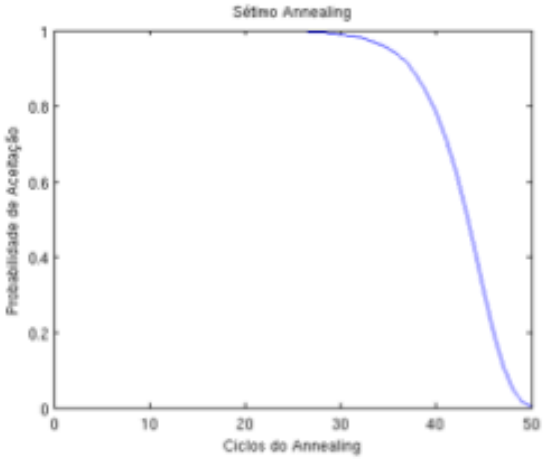
Esta estratégia é boa quando se tem limites de tempo pois ela em um tempo curto deve encontrar bons mínimos na vizinhança da solução inicial e dado um tempo adicional o algoritimo se torna mais agressivo a buscar soluções fora da vizinhança.

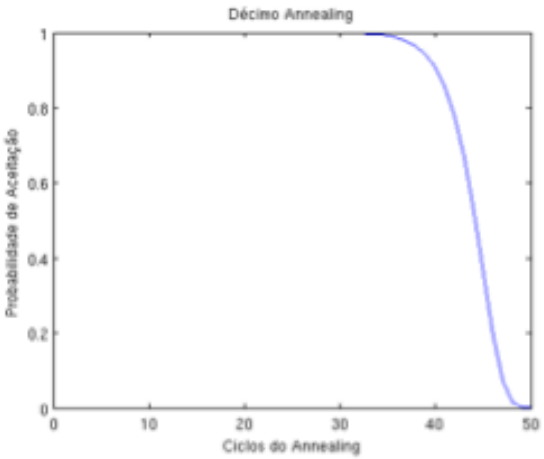
Descrição da escolha de valores:
Utilizamos k=1, tendo em vista que apenas uma constante linear não altera significativamente o comportamento da aleatoriedade. A constante T=1 inicialmente e assume valor 10^n no enésimo annealing rodado. O decaimento de temperatura alfa inicialmente é de 0.99 e ele é alterado 0.95 por annealing rodado. A escolha do s limites de execução de 50 e 150 foram feita afim de que o algoritmo rodasse por volta de 10 vezes em 1 minuto nas máquinas testadas.

Comportamento da Aceitação de Soluções:









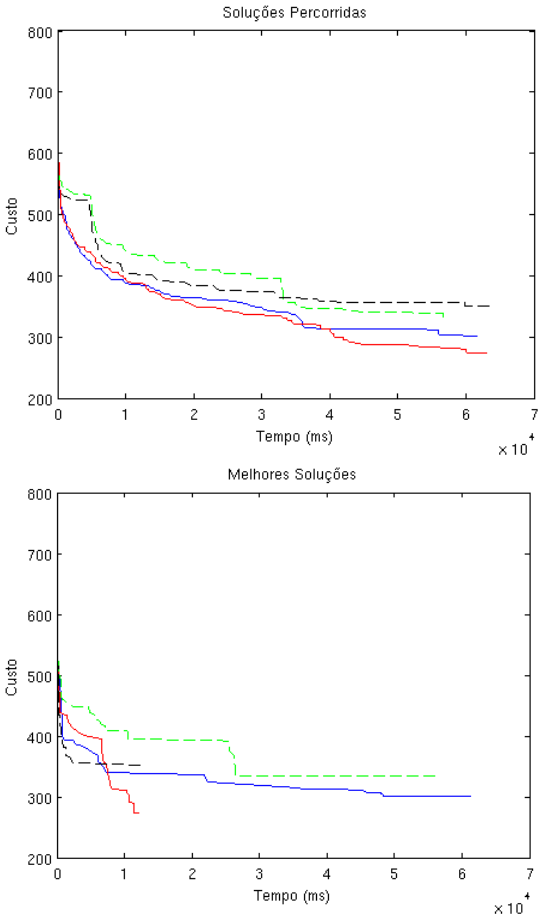
4. Resultados

4.1. **Testes Realizados**

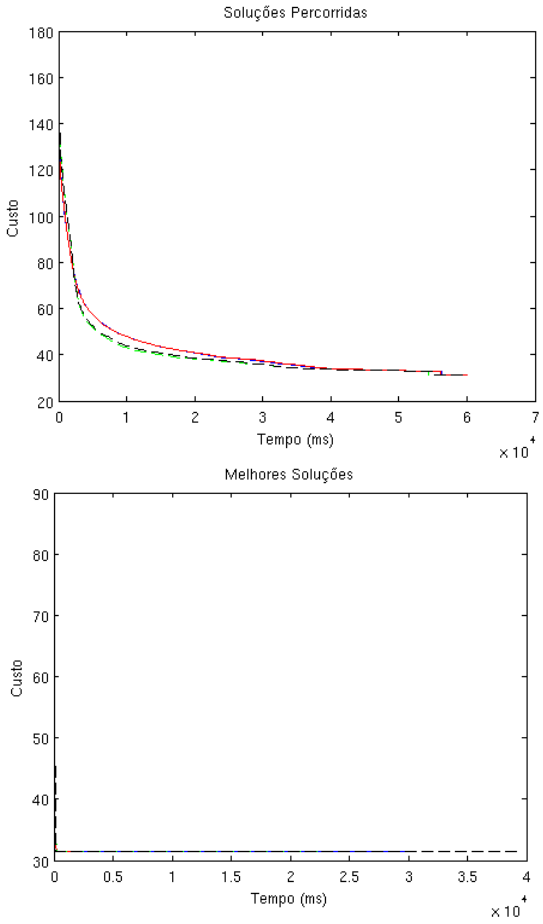
Foram rodadas 2 vezes uma instância de 5 mil pontos para cada método de gerar vizinhança e o mesmo foi feito para a instância de teste fornecida pelo professor.

4.2. **Soluções Encontradas**

Instância de 5 mil pontos:



Instância teste:



Instância 5000 gerador de vizinho aleatório 1

	Linha Azul	Linha Vermelha
Número de execuções:	14	14
Valor:	300.76828440658005	273.9929858347
Total:	42	37

Instância 5000 gerador de vizinho aleatório 2 (tracejado)

	Linha Verde	Linha Preta
Número de execuções:	13	14
Valor:	333.45408228971	351.31435199992995
Total:	39	38

Instância teste gerador de vizinho aleatório 1

	Linha Azul	Linha Vermelha
Número de execuções:	3316	3270
Valor:	31.439873167939982	31.439873167939993
Total:	4	4

Instância teste gerador de vizinho aleatório 2 (tracejado)

	Linha Verde	Linha Preta
Número de execuções:	2319	2256
Valor:	31.43987316793999	31.439873167939986
Total:	4	4

5. Conclusão

O gerador de vizinhança 2 se mostrou menos eficaz que o gerador 1, isso pode ser visto pela instancia de 5mil. A estratégia adotada parece resolver significativamente bem o problema para o tempo de um minuto.

Não foi vista nenhuma grande piora nas soluções correntes o que indica talvez que não haja variação significante dentro de uma vizinhança. Ainda, os graficos das soluções correntes indicam muita diferença ente os comportamentos dos dois geradores adotados.

Faltaria para este trabalho uma comparação da execução rodando em diferentes tempos e comparação com os valores ótimos, este porém desconhecidos no momento.

Como o algoritmo fica mais agressivo com o decorrer da execução e as respostas pouco variavam em custo durante o final das execuções é razoável crer que chegamos em uma resposta consideravelmente próximas do custo ótimo.