TP1: Algoritmo Genético para o problema da p-Mediana com restrições de capacidade

Arthur Câmara Vieira de Souza
arthurcamara@gmail.com

Computação Natural - 2/2012
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)

Resumo. *Este documento tem como propósito apresentar, discutir e analisar o problema da p-Mediana com restrições de capacidade e a aplicação de algoritmos genéticos para sua resolução. São explicados aspectos de modelagem do problema, a implementação das estruturas de dados, algoritmos, instruções de execução e análise de testes e resultados.*

1.INTRODUÇÃO

O problema de localização conhecido como p-Mediana consiste em determinar p centros em um grafo completo, que se conectam aos outros vértices, de forma que cada vértice do grafo esteja conectado a um e exatamente um centro, ou seja, ele próprio um dos p centros. Cada vértice contudo, possui uma demanda e uma capacidade. O nó centro deve ter capacidade suficiente para suprir a sua demanda somada às demandas de todos os vértices com os quais se liga. A solução deve ter exatamente p medianas e a solução ótima é a que minimiza as somas das distâncias entre vértices e centros.

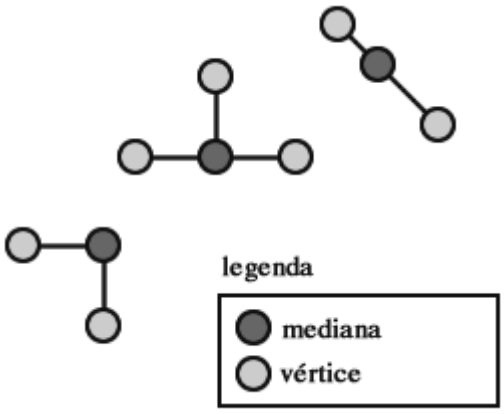

Figura 1. *Exemplo de uma solução ótima de um problema de p-Mediana com p = 3*

O objetivo deste trabalho é resolver o problema da p-Mediana utilizando um algoritmo genético. Desta forma, busca-se uma solução que se aproxime da solução ótima do problema através da evolução de uma população de soluções possíveis no espaço de buscas do problema.

A **Seção 2** deste documento discute a modelagem básica dos indivíduos e do algoritmo proposto. Em seguida, na **Seção 3**, são expostas as decisões de implementação, classes e estruturas de dados implementadas na resolução do problema. Na **Seção 4** foram listados os arquivos do programa e registradas instruções de execução e compilação. A **Seção 5** apresenta os experimentos realizados, bem como a metolodogia adotada. Os resultados e análises podem ser encontrados na **Seção 6**. Finalmente, a **Seção 7** conclui o documento.

2. MODELAGEM BÁSICA E SOLUÇÃO PROPOSTA

2.1 Observações Iniciais

Um problema inicial observado neste trabalho é a quantidade de restrições impostas às soluções. Em geral, são elas:

1. Todos os vértices de uma solução devem estar ligados a pelo menos uma mediana, ou ser uma mediana;
2. Devem existir exatamente p medianas;
3. As medianas devem ter capacidade suficiente para suprir as suas demandas somadas às demandas de seus vértices conectados.

É claro perceber que a geração aleatória de indivíduos pode gerar um grande número de indivíduos inválidos, e operações genéticas entre indivíduos válidos podem gerar indivíduos inválidos facilmente. Isso foi um desafio importante na modelagem do problema e motivo principal da escolha da representação do indivíduo, porque foi considerado importante que as características restritivas do problema sejam mantidas durante o processo de evolução.

2.2 Indivíduo

Um indivíduo neste problema representa uma solução válida para o problema, ou seja, uma configuração que permite representar os centros e suas ligações com outros vértices. Um indivíduo é, basicamente, representado por dois vetores, ***grupos*** e ***centros***.

O primeiro passo na representação de indivíduos foi a divisão do conjunto de vértices em grupos. Cada vértice então, faz parte de um grupo de vértices. Isso é representado, no indivíduo, pelo vetor ***grupos***. Neste, a chave (posição) no vetor representa o vértice, e o valor representa o grupo do qual o vértice pertence. O valor do grupo é um número n, onde $0<n<=p$ e p é o número de medianas (centro).

O indivíduo possui, ainda, um vetor **centros**. Neste vetor, de tamanho igual a p, a chave representa o grupo e o valor representa o elemento do grupo que será considerado mediana.

Com estes vetores, é possível representar soluções em termos de grupos de vértices e seus centros. Uma característica dessa representação é que torna-se mais fácil alterar o centro de um mesmo grupo de vértices. Isso pode ser importante quando se tem uma solução que mantém grupos geograficamente conglomerados, mas que possui um centro ruim como escolha. Esta representação também facilita a mudança de vértices de um grupo para outro, o

que pode ser útil para o refinamento de boas soluções, por exemplo.
A seguir, um exemplo de representação de um indivíduo.

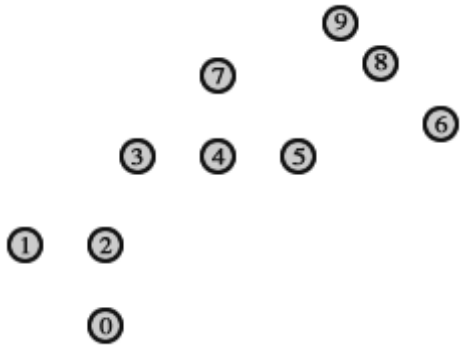

Figura 2. *Conjunto de Vértices*

Tomando os vértices da **Figura 2** como entrada para o problema onde $p = 3$, o vetor **grupos** indivíduo poderia ser representado geneticamente como um vetor da seguinte forma:

```
grupos = [0,1,0,2,1,2,0,0,2,1]
```

Esta representação genética pode ser mapeada para o fenótipo ilustrado pela **Figura 3**.

Figura 3. *Divisão de grupos*

Já vetor **centros** pode ser representado por um vetor como o exemplo:

```
centros = [1,1,2]
```

Este vetor informa que o centro do grupo 0 é o segundo elemento do grupo, o centro do grupo 1 é o segundo elemento do grupo e o centro do grupo 2 é o terceiro elemento do grupo. Com esta representação genética, em grupos e centros, podemos representar o indivíduo pelo fenótipo ilustrado na **Figura 4**.

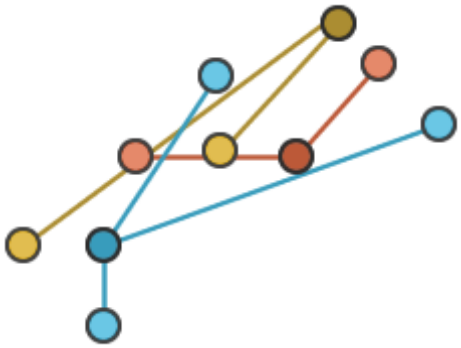

Figura 4. *Fenótipo do indivíduo*

A modelagem do indivíduo dessa forma facilita ainda operações genéticas, uma vez que é possível facilmente recombinar genes (Crossover), o que traz uma solução com vértices de pais em grupos recombinados, e realizar mutações, que seria equivalente a alterar um vértice de grupo ou o centro do grupo.

2.3 A validação de um indivíduo.

Como pode ser percebido pela própria natureza do problema, as operações genéticas e geração espontânea de indivíduos potencialmente geram um número extremamente alto de soluções inválidas. Por isso, ao gerar um indivíduo qualquer, é realizada uma série de três operações de conserto do indivíduo, caso este seja inválido, nesta ordem:

1. Rearranjo dos grupos do indivíduo, de forma a garantir que este indivíduo possui exatamente p grupos e que cada grupo tenha ao menos 2 vértices (um centro e um vértice, no mínimo). Essa operação garante que as restrições 1 e 2 (ver item 2.1) sejam obedecidas.
2. Sorteio de novo centro, caso necessário, de forma a buscar atender à restrição 3 (ver item 2.1)
3. Caso ainda seja inválido, alteração de um vértice do grupo problemático para outro grupo, e reavaliação de validade do indivíduo.

Ao mesmo tempo em que a correção pode ser ruim na medida em que pode reduzir o espaço de busca do algoritmo, o simples descarte do indivíduo provou, experimentalmente, ser terrível em termos de desempenho. Muitas vezes um indivíduo bom se torna inválido por problemas "pequenos" e fáceis de corrigir, como o isolamento de um vértice único em um grupo, ou a escolha de um centro ruim para um grupo. Espera-se que a mutação, a correção aleatória e as operações genéticas possam introduzir a inovação e diminuir o impacto da correção de indivíduos na convergência do algoritmo.

2.3 O valor da Fitness

A *Fitness* de um indivíduo é dada pela soma das distâncias de cada centro aos seus vértices. A escolha foi feita de forma a estar diretamente relacionada ao objetivo do problema, que é minimizar a soma das distâncias. Assim, quanto menor a *fitness* de um indivíduo, melhor este

indivíduo é. Considerando p o número de medianas (grupos) da minha solução, n o número de vértices de um dado grupo i, v_x^{ij} como a posição horizontal do vértice v^{ij}, v_y^{ij} como a posição vertical do vértice v^{ij}, c_x^i como a posicão horizontal do centro c^i e c_y^i como a posicão vertical do centro c^i, e temos que a *fitness* de um indivíduo I é dada pela função f abaixo:

$$f(I) = \sum_{i=0}^{p} \sum_{j=0}^{n} \sqrt{(v_x^{ij} - c_x^i)^2 + (v_y^{ij} - c_y^i)^2}$$

2.4 Crossover e Mutação

2.4.1 Crossover

A operação de crossover de dois indivíduos gera, como esperado, novos dois indivíduos. Para tal, a operação de crossover leva em conta duas chaves, aleatoriamente geradas $c1$ e $c2$ que representam os pontos de corte nos vetores ***grupos*** e ***centros***, respectivamente, dos dois indivíduos.

Exemplo de CrossOver entre indivíduos $ind1$ e $ind2$, com $c1 = 4$ e $c2 = 1$:

```
ind1 { grupos = [0,1,0,2|1,2,0,0,2,1] centros = [1|1,2] }
ind2 { grupos = [1,1,0,0|0,0,2,0,2,1] centros = [0|1,0] }
```

Após CrossOver:

```
ind3 { grupos = [0,1,0,2,0,0,2,0,2,1] centros = [1,1,0] }
ind4 { grupos = [1,1,0,0,1,2,0,0,2,1] centros = [0,1,2] }
```

2.4.2 Mutação

A mutação leva em conta uma probabilidade de mutação pM para mutar cada gene dos vetores ***grupos*** e ***centros***, de cada um dos pais. Mutar, neste contexto, significa gerar um novo valor aleatório para o item do vetor.

Exemplo de Mutação nos indivíduos $ind1$ e $ind2$, com itens a serem mutados em destaque:

```
ind1 { grupos = [0,1,0,2,1,2,0,0,2,1] centros = [1,1,2] }
ind2 { grupos = [1,1,0,0,0,0,2,0,2,1] centros = [0,1,0] }
```

Após Mutação:

```
ind3 { grupos = [0,2,0,2,1,2,0,1,2,1] centros = [1,1,2] }
ind4 { grupos = [1,1,2,1,0,0,0,0,2,1] centros = [0,1,2] }
```

2.5 O algoritmo genético

O algoritmo proposto é relativamente simples, mas possui alguns pontos importantes a serem considerados. O algoritmo pode ser resumido no conjunto de passos a seguir:

1. Uma população inicial é gerada, com indivíduos totalmente válidos (ver 2.1 e 2.2).
2. Os indivíduos são mapeados em soluções (fenótipo) e o valor de *fitness* é obtido, calculando a somas das distâncias de cada centro aos seus vértices (ver 2.3).
3. Os indivíduos passam seleção por torneio e uma nova população de indivíduos é gerada, em que indivíduo é gerado por mutação ou crossover (ver 2.4).
4. A nova população substitui totalmente a antiga (exceto se elitismo for aplicado, quando 1 ou mais pais passam para a próxima geração).
5. Repete-se os passos 2 a 4 até que o número de gerações desejada seja alcançado.
6. A solução encontrada é o melhor indivíduo da última geração.

O algoritmo implementado é bem tradicional. Uma das coisas que o tornam menos tradicional é o fato de ocorrerem as correções para validação dos indivíduos, como explicado anteriormente.

A solução implementada leva em conta ainda alguns parâmetros, como tamanho da população inicial, número de gerações, tamanho do torneio, elitismo e probabilidades de crossover e mutação. Na **Seção 4** é possível verificar como executar o programa com inserção destes parâmetros.

3. IMPLEMENTAÇÃO

A implementação da solução proposta foi feita em C++. Foram criados alguns objetos principais que representam os modelos criados. Cada objeto é implementado como uma classe, e possui campos e métodos. Os principais campos e métodos serão abordados a seguir. Todos os campos e métodos podem ser verificados no código-fonte do programa, disponibilizado em conjunto com esta documentação.

3.1 Individuo

A classe *Individuo* representa um indivíduo conforme descrito no item 2.2.

3.1.1 Campos importantes de *Individuo*

- **grupos:** vetor de grupos é parte do genótipo do indivíduo.
- **centros:** vetor de centros é parte do genótipo do indivíduo.
- **fitness:** armazena a fitness de um *indivíduo*.

3.1.2 Alguns métodos importantes de *Individuo*

- **geraIndividuo():** gera um Indivíduo aleatório (para a população inicial).

- **crossover(*ind2, c1, c2*):** realiza crossover com outro indivíduo, recebendo também os pontos de corte como parâmetros.
- **mutacao(*prob*):** gera indivíduo a partir mutação com probabilidade *prob* para cada gene.
- **redistribuiGrupos():** torna o indivíduo válido, caso não seja.

3.2 Vertice

A classe *Vertice* representa um vértice do problema.

3.2.1 Campos importantes de *Vertice*

- **demanda:** armazena a demanda do vértice;
- **capacidade:** armazena a capacidade de fornecimento do vértice;
- **pos_x:** armazena a posicao horizontal do vértice;
- **pos_y:** armazena a posicao vertical do vértice;

3.2.2 Alguns métodos importantes de *Vertice*

- **distanciaVertice(vertice2):** Calcula distância do vértice até o *vertice2* recebido como parâmetro.

3.3 Geracao

A classe *Geracao* representa uma geração do algoritmo genético e realiza o processamento principal do algoritmo, como a evolução. Também faz o mapeamento de indivíduos para soluções.

3.3.1 Campos importantes de *Geracao*

- **individuos:** lista de indivíduos do tipo *Individuo* (população desta geração)
- **vertices:** lista de vértices do tipo *Vertice*
- ***outros*:** *são armazenados também outros dados importantes de execução, como número de indivíduos melhores/piores/repetidos, fitness média/melhor/pior, e parâmetros do algoritmo evolucionário, como tamanho do torneio, elitismo, probabilidades, etc.*

3.3.2 Alguns métodos importantes de *Geracao*

- **construtor(*populacao, vertices, num_medianas*):** Cria uma geração e população inicial randômica.
- **avaliarFitnessIndividuo(ind):** com base nos vértices, decodifica o indivíduo em uma solução e avalia a fitness de um indivíduo de sua população, contabilizando também estatísticas importantes;
- **torneio():** realiza torneio retornando o vencedor (melhor indivíduo do torneio).
- **retornaElite():** retorna o(s) melhor(es) indivíduo(s) da população.
- **solucao():** imprime a melhor solução da população da geração.
- **evoluir():** realiza os passos básicos de uma evolução, transformando a geração corrente na seguinte.

3.3.2.1 Pseudo-algoritmo do método evoluir()

```
evoluir() {
      geracao = geracao + 1;
      reinicia_dados_estatisticos();
      nova_populacao = [];
      if(elitismo) nova_populacao.inserir(retornaElite());
      while(nova_populacao.size() < populacao_corrente.size()) {
            escolhe pai1 e pai2 por torneio(); //diferentes

            //crossover ou mutacao?
            if(crossover && pai1 != pai2) {
                  c1 = PontoCorte();
                  c2 = PontoCorte();
                  filho1 = pai1.crossover(pai2, c1, c2);
                  filho2 = pai2.crossover(pai1, c1, c2);
            }
            else {
                  filho1 = pai1.mutacao();
                  filho2 = pai2.mutacao();
            }

            if(filho1 INVALIDO) filho1.corrigeIndividuo();
            if(filho2 INVALIDO) filho1.corrigeIndividuo();

            Contabiliza dados estatísticos;

            nova_populacao.inserir(filho1);
            nova_populacao.inserir(filho2);
      }
      populacao = nova_populacao; //nova populacao substitui antiga
}
```

3.4 Considerações sobre execução

A execução da evolução de fato, consiste, portanto, na criação de uma geração inicial (população inicial) randômica e na chamada repetitiva do método *evoluir()* para obter gerações seguintes.

3.3 Outras decisões de implementação

Para melhorar o desempenho do algoritmo e melhorar o desempenho do cálculo das distâncias, todos os vértices são movidos para próximo da origem no plano cartesiano. Isso ocorre porque não importa para o problema, de fato, a posição absoluta dos vértices, mas sim a posição relativa de cada um deles. A **Figura 5** ilustra este ajuste, feito num primeiro momento, quando ocorre a leitura dos dados de entrada do problema.

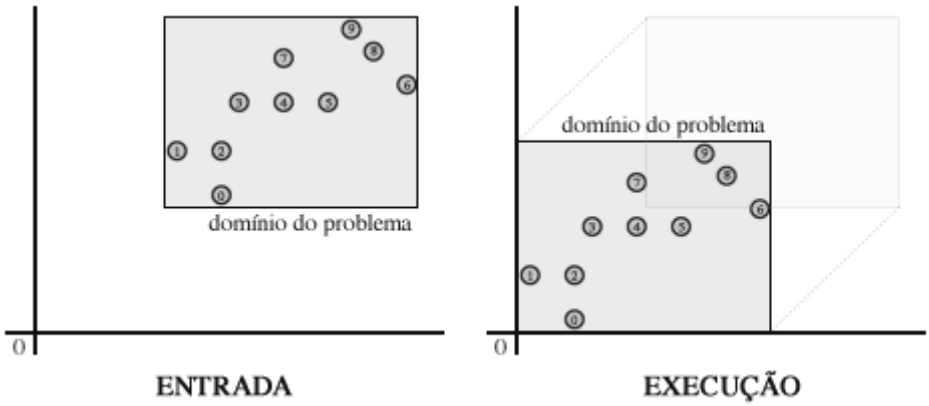

Figura 5. *Vértices movidos para a origem na execução*

O cálculo da *fitness*, por exemplo, é beneficiado por essa medida, porque leva em conta a distância de pontos. Como o cálculo da distância executa operações como exponenciação, valores menores tendem a melhorar o desempenho geral do algoritmo e necessitam de menos memória.

4. ARQUIVOS, COMPILAÇÃO E EXECUÇÃO

O programa foi implementado em C++. Os arquivos de código-fonte do programa se encontram na pasta *src* no pacote disponibilizado desta documentação.

4.1 Arquivos

São partes do código-fonte do programa os seguintes arquivos:

- **main.cpp:** arquivo principal de execução
- **config.cpp:** arquivo de configuração básica e definição de constantes e valores padrões para parâmetros
- **lib.cpp:** funções comuns úteis
- **Individuo.cpp e Individuo.h:** arquivos fonte e header da classe *Individuo*
- **Vertice.cpp e Vertice.h:** arquivos fonte e header da classe *Vertice*
- **Geracao.cpp e Geracao.h:** arquivos fonte e header da classe *Geracao*

4.2 Compilação

A compilação pode ser feita através do Makefile disponível na pasta *src*. Para isso, acesse a pasta *src* do pacote via terminal e digite o comando:

```
make
```

Alternativamente, é possível compilar com o comando a seguir, a partir da pasta *src*:

```
g++ main.cpp individuo.cpp vertice.cpp geracao.cpp lib.cpp -o ../
pmedianas
```

4.4 Execução

Para executar o programa, navegue até a pasta raiz pelo terminal e digite o comando da forma:

```
./pmedianas [parametros, ...] < [arquivo_entrada]
```

Os parâmetros são opcionais, e assumirão o valor padrão se não forem informados. Para enviar parâmetros, estes devem ser enviados nesta ordem:

1. População Inicial (inteiro). Padrão: 1000
2. Número de Gerações (inteiro). Padrão: 1000
3. Semente para geração de população inicial (inteiro). Padrão: 0 (0 = semente aleatória)
4. Semente para evolução (inteiro). Padrão: 0 (0 = semente aleatória)
5. Tamanho do Torneio (inteiro). Padrão: 2
6. Tamanho do Elitismo (inteiro). Padrão: 0 (nenhum elitismo)
7. Probabilidade de CrossOver (float). Padrão: 0.6
8. Probabilidade de Mutação (float). Padrão: 0.001
9. Passos para imprimir gerações (inteiro). Padrão: 1 (de 1 em 1)

Exemplos de chamadas de execução com parâmetros:

1. População inicial de 100, 50 gerações, outros valores padrões, e arquivo de entrada *teste.db*:
   ```
   ./pmedianas 100 50 < teste.db
   ```

2. População inicial de 1000, 3000 gerações, sementes 4, torneio de tamanho 10, outros valores padrões, e arquivo de entrada *teste.db*:
   ```
   ./pmedianas 1000 3000 4 4 10 < teste.db
   ```

3. População inicial de 10000, 1000 gerações, sementes 3, torneio de tamanho 7, elitismo de 2 indivíduos, probabilidade de crossover 0.8, probabilidade de mutação 0.05, imprimindo de 20 em 20 gerações e arquivo de entrada *teste.db*:
   ```
   ./pmedianas 10000 1000 3 3 7 2 0.8 0.05 20 < teste.db
   ```

4.4.1 Entrada

O arquivo de entrada deve seguir o seguinte formato:

```
Linha1: n p
i-ésima linha: x y c d
```

Onde *n* é o número de vértices, *p* o número de centros (medianas), e cada linha subsequente representa um vértice, sendo *x* a cordenada *x* do i-ésimo vértice, *y* a cordenada *y* do i-ésimo vértice, *c* a capacidade do vértice e *d* a demanda do mesmo vértice.

4.4.2 Saída

Os resultados das execuções exibem uma série de dados. Para cada geração é impresso:

1. Melhor *fitness* da geração;
2. Pior *fitness* da geração;
3. *Fitness* média da geração;
4. Número de indivíduos melhores que os pais (gerados por crossover);
5. Número de indivíduos piores que os pais (gerados por crossover);
6. Número de indivíduos iguais.

Ao final da execução é exibida ainda a melhor solução encontrada na última geração e o tempo de execução do programa.

5. EXPERIMENTOS

Aqui apresento os experimentos realizados. Os testes foram realizados em um computador com processador 2.5 GHz Intel Core i5, com 4GB de memória DDR3 e sistema operacional Mac OS X 10.7.4.

5.1 Metodologia

Diversos experimentos foram realizados e alguns exemplos deles estão na pasta *tests* junto aos arquivos. Os testes foram automatizados através de scripts *bash* com extensão *.sh*, também disponíveis na pasta *tests*. Nem todos os arquivos resultantes dos testes automatizados estão no pacote do trabalho visando diminuir o tamanho geral do pacote enviado. Para repetir todos os testes executados basta executar o arquivo *tests_main.sh*, disponível na pasta *tests*, com o comando no terminal:

```
./tests_main.sh
```

Alguns exemplos de saídas de testes, com todas as bases de dados, encontram-se em *tests/ exemplos_saidas/*.

5.2 Experimentos

Abaixo estão descritos os experimentos realizados. Os resultados e discussões de cada experimento estão na **Seção 6**.

5.2.1 Experimento 1: Convergência da população

Neste primeiro experimento, o objetivo é simplesmente verificar se o algoritmo caminha para uma boa solução para diferentes entradas do problema da p-Mediana. Este experimento pode ser executado individualmente executando o script tests/test_conv.sh.

Parâmetros dos experimentos:

População Inicial = 100
Número de Gerações = 500
Tamanho do Torneio = 2
Tamanho do Elitismo = 0

Probabilidade de CrossOver = 0.6
Probabilidade de Mutação = 0.001

Para cada configuração possível foram realizados 30 testes, e a média foi considerada.

5.2.2 Experimento 2: Variação do tamanho da população e número de gerações

Neste experimento, o objetivo é avaliar parâmetros para o tamanho da população inicial e número de gerações. Este experimento pode ser executado individualmente executando o script tests/test_pop_ger.sh.

Parâmetros dos experimentos:
População Inicial = $500 \leq p \leq 2500$
Número de Gerações = $500 \leq g \leq 2500$
Tamanho do Torneio = 2
Tamanho do Elitismo = 1
Probabilidade de CrossOver = 0.6
Probabilidade de Mutação = 0.001

Para cada configuração possível foram realizados 30 testes, e a média entre os testes foi considerada.

5.2.3 Experimento 3: Variação das probabilidades

Neste experimento, o número de indivíduos da população inicial e o número de gerações foi fixado com base no Experimento 2. O objetivo, neste momento, é descobrir melhores parâmetros para as probabilidades de Crossover e Mutação. Este experimento pode ser executado individualmente executando o script tests/test_prob.sh.

Parâmetros dos experimentos:
População Inicial = 1.500
Número de Gerações = 1.500
Tamanho do Torneio = 2
Tamanho do Elitismo = 1
Probabilidade de CrossOver = $0.3 \leq p_c \leq 0.99$
Probabilidade de Mutação = $0.0001 \leq p_m \leq 0.01$

Para cada configuração possível foram realizados 30 testes e a média entre os testes foi considerada.

5.2.4 Experimento 4: Torneio

Com base nos experimentos anteriores, puderam ser estimados bons valores para os parâmetros de população inicial e número de gerações, além das probabilidades de cada operação genética. Agora desejamos investigar, com estes mesmos valores, a alteração que o tamanho do torneio provoca. Este experimento pode ser executado individualmente

executando o script tests/test_torneio.sh.

Parâmetros dos experimentos:
População Inicial = 1.500
Número de Gerações = 1.500
Tamanho do Torneio = 2
Tamanho do Elitismo = 1
Probabilidade de CrossOver = 0.6
Probabilidade de Mutação = 0.01

Para cada configuração possível foram realizados 30 testes, com sementes diferentes, e a média entre os testes foi considerada.

5.2.4 Experimento 5: Elitismo

Com base nos experimentos anteriores, puderam ser estimados bons valores para os parâmetros. Este experimento buscou determinar o impacto do elitismo na convergência das soluções, retirando agora o elitismo. Este experimento pode ser executado individualmente executando o script tests/test_elitismo.sh.

Parâmetros dos experimentos:
População Inicial = 1.500
Número de Gerações = 1.500
Tamanho do Torneio = 2
Tamanho do Elitismo = 0
Probabilidade de CrossOver = 0.6
Probabilidade de Mutação = 0.01

Para cada configuração possível foram realizados 30 testes, com sementes diferentes, e a média entre os testes foi considerada.

6. RESULTADOS

6.1 Experimento 1: Convergência da população

Foi constatado que o algoritmo, de fato, evolui a solução de forma que os resultados gerados caminham para uma solução melhor. A **Figura 6** exibe a média do melhor valor de *fitness* entre 30 experimentos idênticos, com cada uma das 3 entradas diferentes. O eixo das ordenadas exibe o melhor fitness encontrado em cada uma das 500 gerações (eixo das abscissas). Neste experimento, podemos ver que a primeira entrada convergiu rapidamente para a solução ótima, enquanto as outras duas entradas não conseguiram alcançar bons resultados com estes parâmetros.

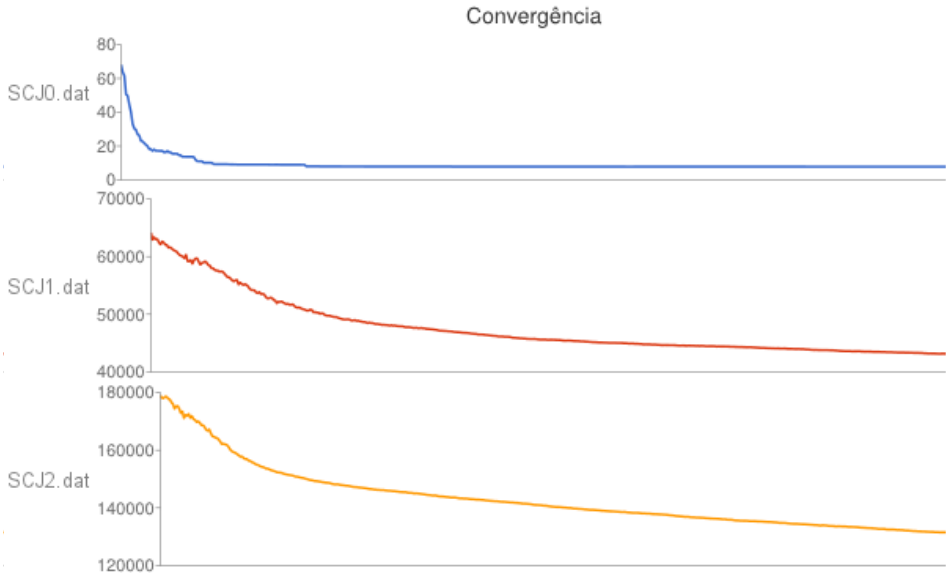

Figura 6. *Melhor fitness melhora com as gerações*

A solução ótima, não atingida, para a entrada SCJ1.dat é 17.246,53 e da entrada SCJ2.dat é 33.225,88. O que podemos concluir é que o algoritmo é capaz de convergir para uma solução boa (como ocorreu em SCJ0.dat), mas os parâmetros não estão adequados para as outras entradas. Ao que podemos observar, houve uma convergência prematura da população dos dois últimos, que acabou prejudicando o resultado final.

6.2 Experimento 2: Tamanho da população e número de gerações

Com a execução e análise do segundo experimento, pudemos encontrar um tamanho de população e número de gerações bons para a maioria dos casos.

O gráfico representado na **Figura 7** abaixo, mostra um exemplo com o arquivo SCJ1.data, com as médias das melhores fitness dos testes para este arquivo. As cores representam as gerações 500, 1000, 1500, 2000 e 2500.

Percebe-se ainda que as populações devem ser maiores quando o problema é mais complexo. Isso se dá porque populações maiores permitem explorar melhor o espaço de busca e geram soluções iniciais mais diversas.

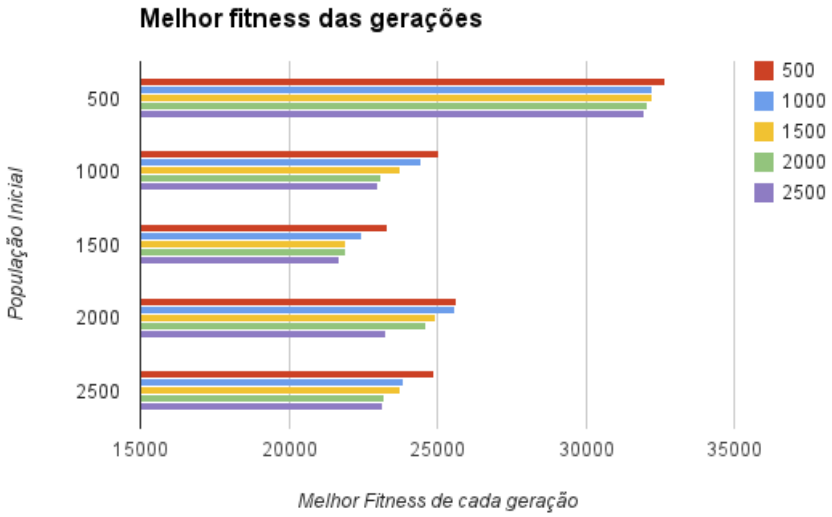

Figura 7. *População Inicial X Gerações - Para entrada SCJ1.data*

Podemos perceber, por exemplo, que o número de indivíduos na população inicial tem um impacto muito importante na qualidade da solução final. Neste experimento percebe-se, por exemplo, que a média de soluções encontradas com uma população inicial de 500 indivíduos não se alterou muito, ainda que o número de gerações tenha sido incrementado consideravelmente. Percebe-se também que o número de gerações não afetou muito a solução final obtida. As melhores fitness nas gerações 1500, 2000 e 5000 são muito próximas. Isso mostra que o algoritmo está convergindo rapidamente, e não está evoluindo muito após a 1500ª geração. Um dado que evidencia isso ainda mais é a média das fitness dessas gerações. Como a Figura 8 mostra, a média nestas gerações é muito próxima da melhor fitness. Isso evidencia que a maioria da população está perto da solução encontrada.

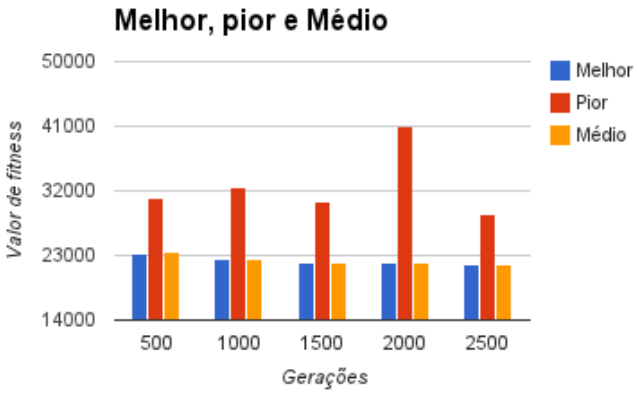

Figura 8. *Melhor, pior e fitness média.*

Neste experimento, podemos perceber que o melhor número obtido para a população inicial foi de 1500. As outras entradas apresentam comportamento muito similar, como pode ser visto no gráfico da entrada SCJ2.data, abaixo:

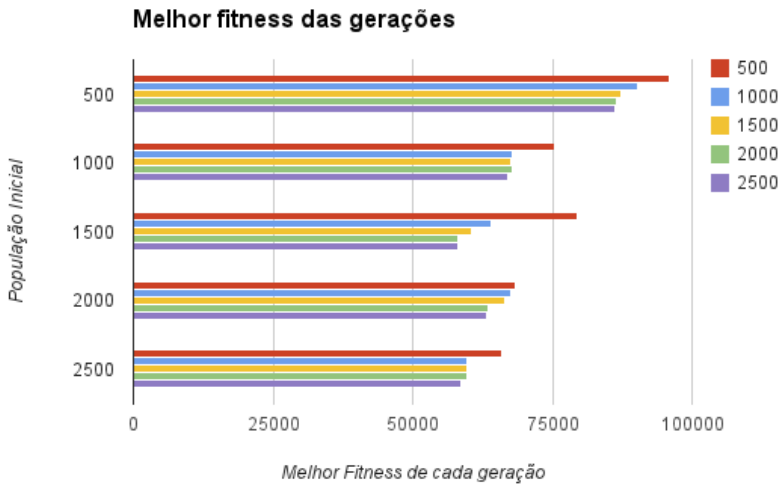

Figura 9. *População Inicial X Gerações - Para entrada SCJ1.data*

6.3 Experimento 3: Probabilidades de Mutação e Crossover

O terceiro experimento ajuda a buscar as melhores probabilidades de mutação e crossover.

Com a realização dos testes do terceiro experimento, ficou claro que o aumento da mutação ajudou a introduzir novidades no espaço de busca. Pode ser percebido, em diversos testes, que o número de indivíduos repetidos é muito alto, e o número de indivíduos melhores nas gerações seguintes tende muito baixo quando o resultado converge para uma solução. O exemplo abaixo ilustra este fato em uma execução qualquer, mas isso pode ser percebido em praticamente todas as execuções. Neste exemplo a população tem tamanho 1000, o que mostra a quantidade de indivíduos iguais após várias gerações

```
Geracao 1000
Melhor Fitness: 67959.1
Pior Fitness:    86849.2
Fitness Médio: 68404.6
Melhores: 6
Piores: 6
Repetidos: 862

Geracao 1500
Melhor Fitness: 67452.3
Pior Fitness:    86018.5
Fitness Médio: 67890.9
Melhores: 7
Piores: 7
Repetidos: 795
```

Figura 10. *Indivíduos iguais em uma instância de população igual a 1000*

Isso ocorre, acredito, porque os indivíduos são corrigidos para serem válidos, e pouco mutados. Com isso, muitos se tornam iguais. O melhor desempenho do algoritmo foi com mutação igual a 0.01, ou 1% dos genes mutados.

Quanto ao crossover, o resultado que prevaleceu foi o de 0.6 de probabilidade de crossover. Resultados muito altos, como 0.99 geraram muitos indivíduos repetidos.

Na maioria dos cenários, a situação ilustrada pelo gráfico da **Figura 11** prevaleceu.

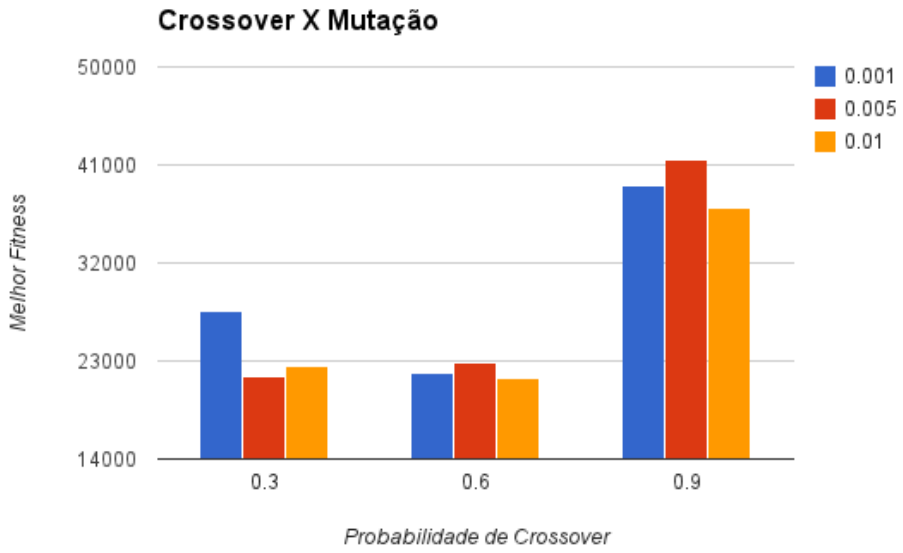

Figura 11. *Melhor situação, neste caso, foi pc = 0.6 e pm = 0.01*

6.4 Experimento 4: Torneio

Os experimentos com torneio mostraram que o aumento do tamanho do torneio para 5 melhoraram muito a solução final. Contudo, o aumento do torneio para mais de 5 não afetaram e, inclusive, em vários testes, pioraram o desempenho.

6.5 Experimento 5: Elitismo

Experimentos com elitismo mostraram que a ausência do elitismo piorou a solução final.

6.6 Resultados Gerais

Mesmo com os parâmetros aprimorados, nenhuma das entradas (exceto SCJ0.dat) conseguiu atingir o valor ótimo no algoritmo genético implementado nestes experimentos. Em especial, as entradas mais complexas, com mais vértices e mais p-medianas, diveram uma aproximação menor da solução final.

Abaixo estão dispostos gráficos que representam médias de testes para cada uma das quatro entradas SCJ1.dat, SCJ2.dat, SCJ3b.dat e SCJ4a.dat. Nos gráficos abaixo, a linha azul representa a solução encontrada pelo algoritmo em cada uma das gerações e a linha verde representa a solução ótima do problema:

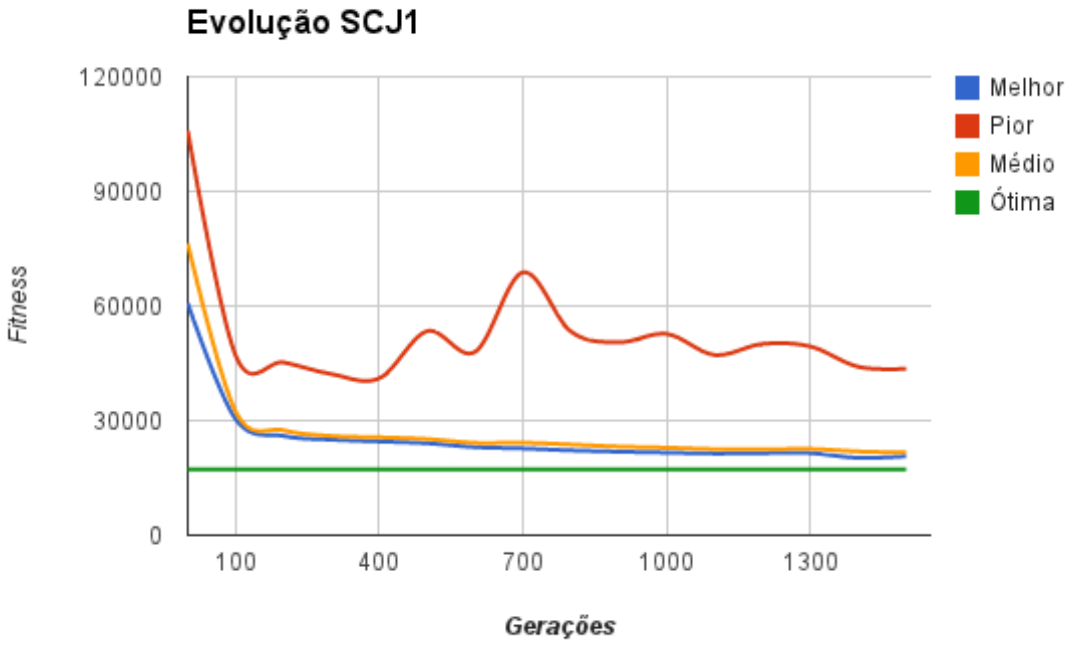

Figura 12. *SCJ1 teve resultado bom com os parâmetros*

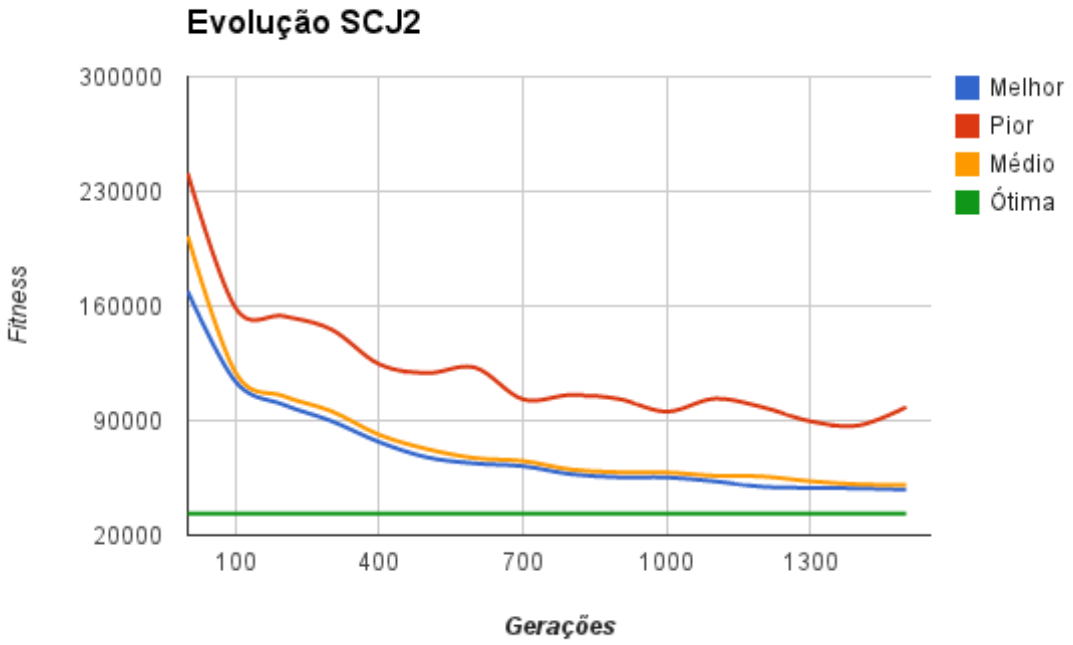

Figura 13. *SCJ2 teve uma convergência mais lenta*

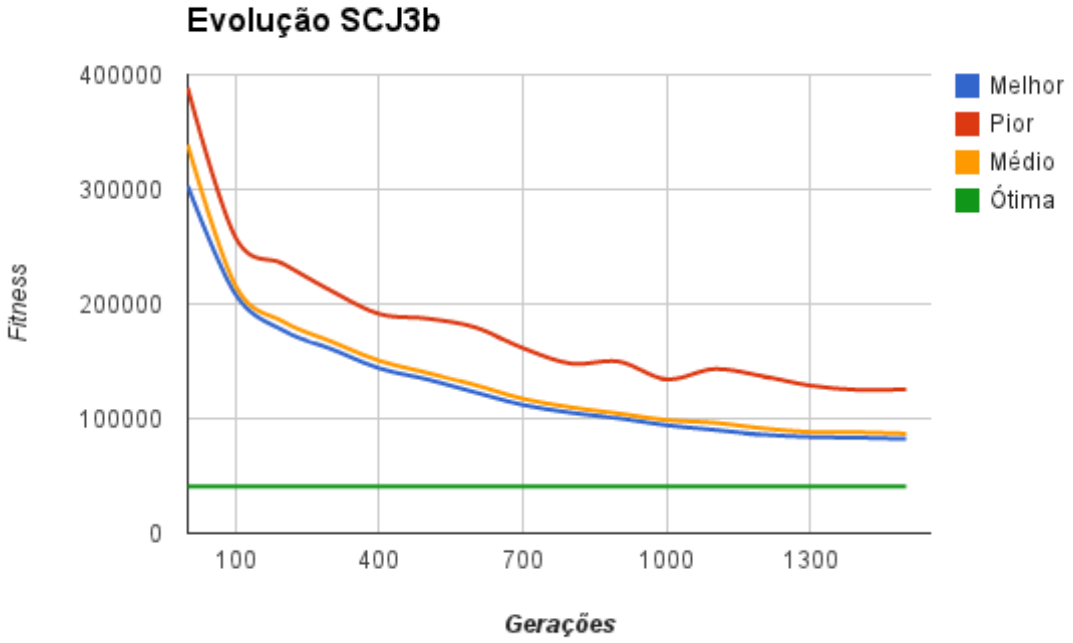

Figura 14. *SCJ3b ficou um pouco distante da solução ótima*

No gráfico da Figura 14, percebemos que para a entrada SCJ3b, o desempenho foi um pouco pior. A solução média encontrada para 1500 gerações foi de 82.224, e para 3000 gerações de 78.000. Considerando que a solução ótima é de, aproximadamente 40636, a solução do algoritmo genético nestes parâmetros ficou um pouco distante.

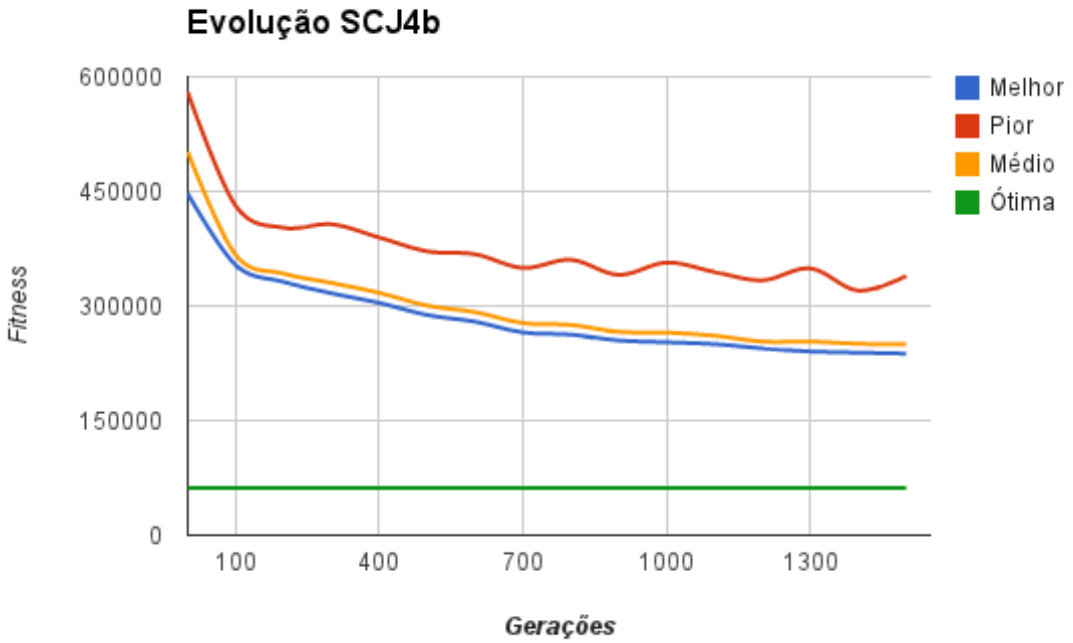

Figura 15. *SCJ4a teve o pior dos desempenhos entre os 4*

7. CONCLUSÃO

É muito interessante perceber o desempenho e evolução das soluções num algoritmo evolucionário, convergendo para a solução. Ao mesmo tempo, a escolha dos melhores parâmetros não é uma tarefa trivial, uma vez que o tamanho da população inicial, a mudança de probabilidades, alteração do tamanho do torneio, ou o uso do elitismo podem alterar drasticamente o resultado final do algoritmo.

Por isso, acredito que a realização de testes que exploram separadamente cada um destes parâmetros foi fundamental para o melhor entendimento do papel de cada um deles na pressão seletiva e convergência da população. Ainda assim, percebe-se que é difícil determinar a combinação exata de parâmetros capaz de prover a solução ótima, mas é possivel estabelecer conjuntos de parâmetros que conferem soluções relativamente boas.

Embora as entradas mais complexas não tenham tido um resultado tão bom quando as mais simples, elas conseguiram evoluir. Talvez o fato de corrigir indivíduos inválidos, por exemplo, possa ter prejudicado estes casos, ou ainda fossem necessários testes com populações ainda maiores para estas instâncias, o que certamente necessitaria de poder computacional muito maior, mas que possivelmente alcançaria melhores resultados.

De qualquer forma, para problemas complexos, como o da p-Mediana com restrição de capacidade o algoritmo genético cumpre um papel importante, pois consegue prover soluções que tendem ao ótimo com sua abordagem evolucionária.