Relatório Técnico: Solução exata para o problema do Caixeiro Viajante (PCV)

Aluno: Luca Samuel Dos Santos
Disciplina: Projeto e Análise de Algoritmos
Professor: Douglas Donizeti de Castilho Braz

Resumo

Este projeto apresenta o desenvolvimento da solução do problema do Caixeiro Viajante de forma exata. Dado a natureza NP-Completo do problema, para alcançar uma solução exata para no mínimo 26 cidades em tempo de execução de no máximo 10 minutos, foi implementado um algoritmo Branch and Bound (Ramificação e poda). Para maximizar a eficiência das podas, o algoritmo utiliza uma abordagem híbrida, para o limite superior (Upper Bound) foi utilizado o resultado obtido via meta-heurística (Algoritmo Genético) e para o limite inferior (Lower Bound) foi calculado dinamicamente utilizando o algoritmo da árvore geradora mínima (Kruskal).

Fundamentação Teórica

O problema consiste em encontrar o ciclo Hamiltoniano de menor custo em um grafo ponderado, visitando cada vértice exatamente uma vez e retornando à origem. A complexidade de tempo para uma abordagem ingênua é O(n!).

Branch and Bound

O método Branch and Bound organiza o espaço de busca em uma árvore de estados. A cada passo recursivo o algoritmo “escolhe” o próximo vértice a visitar. Se a estimativa de custo de um caminho parcial já excede o melhor custo conhecido, esse ramo inteiro da árvore é descartado. Essas técnicas permitem que o algoritmo seja capaz de chegar à solução exata para entradas de até 26 cidades, em piores casos, que é quando os valores dos pesos das arestas são muito próximos e o algoritmo não é capaz de podar muitos caminhos precocemente.

Estimativa de Kruskal (AGM)

Para decidir se um ramo deve ser podado, é necessário calcular qual seria o custo mínimo para completar o caminho atual. Para isso, utiliza-se o algoritmo de Kruskal para calcular a árvore geradora mínima dos vértices não visitados.

Embora uma árvore não seja um ciclo (relaxamento do problema), o custo da AGM + as arestas de conexão (ida e volta) fornece um limite inferior matemático rigoroso. Se o custo atual + AGM for maior ou igual ao custo da melhor solução, é matematicamente impossível que aquele caminho melhore o resultado, permitindo a poda segura.

Detalhes da implementação e Estruturas de Dados

A eficiência do algoritmo depende drasticamente da velocidade de acesso aos dados. Por isso, para evitar cálculos e ordenações desnecessárias e melhorar o acesso aos dados foi implementado as seguintes otimizações ao algoritmo:

Matriz de adjacências e Cache

Embora listas de adjacências economizem memória, o acesso a matrizes é O(1), o que é essencial para a otimização do algoritmo para alcançar as 26 cidades. Foi criada uma estrutura de cache (cachePesos) para evitar a sobrecarga de chamadas de métodos de objetos (getPeso()) durante a recursão profunda.

Ordenação Prévia

No início da execução do algoritmo, para cada vértice, seus vizinhos são ordenados por peso de aresta. Pois ao tentar acessar os vizinhos mais próximos primeiro, o algoritmo tende a encontrar boas soluções (limite superior baixo) mais rápido, o que aumenta a taxa de poda nos estágios iniciais.

Union Find para Kruskal

Foi implementada a estrutura de dados Union-Find (Conjuntos Disjuntos) com compressão de caminhos. Isso permite verificar a formação de ciclos e unir componentes conexos em tempo quase constante durante o cálculo do limite inferior.

Integração Híbrida

O algoritmo começa a busca apenas após executar o Algoritmo Genético para encontrar o melhor limite superior possível. Isso evita que o algoritmo perca tempo explorando soluções ruins no início da execução.

Algoritmo Genético

O algoritmo genético resolve o problema de maneira aproximada sem garantia da otimalidade, porém, de maneira muito rápida, o que é perfeito para ser usado como limite superior no algoritmo ótimo, pois o mesmo é capaz de chegar próximo à resposta exata, podando muitos “galhos” de uma vez.

Para criação do algoritmo genético foi utilizado o pseudo-código seguinte:

```
BEGIN
  INITIALISE population with random candidate solutions;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    1 SELECT parents;
    2 RECOMBINE pairs of parents;
    3 MUTATE the resulting offspring;
    4 EVALUATE new candidates;
    5 SELECT individuals for the next generation;
  OD
END
```

Inicializar População

Para inicializar a população, foi calculada uma heurística dos vizinhos com menores pesos e inserido na primeira posição da matriz de indivíduos. O restante da população foi embaralhada mantendo o primeiro “gene” igual para todos os indivíduos e garantindo que não houvesse cidades duplicadas.

Avaliar Candidato

Para avaliar os candidatos, foi calculado o fitness de cada um deles (o quão bom é esse candidato). Para candidatos os quais não haviam caminho existente válido para o problema, foi atribuído o peso máximo fazendo com que seu fitness fosse zero, porém mantendo os mesmos para que houvesse material genético para chegar a uma solução boa ao final da execução.

Condição

Para condição de término de execução, foi determinado um número fixo de gerações que pode ser alterado no código para calibrar o algoritmo.

Selecionar os pais

Para selecionar os pais, foi implementada duas soluções, podendo escolher qual delas será usada para calibração do algoritmo através da variável int tipoSelecao (1 = roleta, 2 = torneio). Uma delas é a escolha por Roleta, no qual o código escolhe os candidatos com base nos valores acumulados de seus Fitness dando uma maior chance para pais que possuem um maior fitness. A outra solução é a escolha por Torneio no qual os pais são escolhidos aleatoriamente com base no melhor fitness encontrado até o momento, por exemplo, sorteia um valor, armazena o fitness do mesmo como melhor até o momento e repete o processo até o valor determinado pela variável int tamanhoTorneio.

Recombinar pais

Para o processo de recombinar os pais, após ter selecionado ambos os pais, é sorteado dois pontos de corte em um pai e inserido no filho, seguindo as cidades do segundo pai, evitando sempre a duplicidade de cidades.

Mutação

Para a mutação, o indivíduo pode ou não sofrer mutação a depender da “sorte”, a chance de um indivíduo sofrer mutação depende da variável double taxaDeMutacao, no qual pode ser alterada para calibração do resultado. Foi implementado quatro tipos de mutação: Mutação por Inserção, mutação por troca, mutação por inversão e mutação por mistura. Podendo escolher entre uma dessas mutações para execução a depender da variável int tipoMutacao (1 = inserção, 2 = troca, 3 = inversão, 4 = mistura).

Finalização

Para finalizar a execução, os filhos gerados são avaliados e recebem seus respectivos fitness. E ao final do algoritmo é decidido os indivíduos que seguirão para a próxima geração. Caso a variável boolean elitismo esteja definida como ‘True’, o melhor indivíduo

daquela população será mantido e o resto será trocado pelos filhos, caso contrário, todos os indivíduos são trocados.

Resultados obtidos no AG

Realizando vários testes de calibração, foi possível chegar na solução exata na qual é obtida frequentemente. O resultado pode não ser o exato sempre, porém realizando os testes, chegou-se ao resultado correto em 15, dos 15 testes realizados, portanto pode-se dizer que a calibração ficou boa. O resultado ótimo foi verificado para uma população na qual é fácil obter o melhor resultado pois é possível podar muitos galhos para facilitar os testes. O algoritmo genético é perfeito para entradas muito grandes, porém é importante evidenciar que a resposta pode não ser a exata, mas com um algoritmo bem calibrado, o AG é capaz de chegar bem próximo à resposta exata, ou até mesmo, à resposta exata.

Resposta exata gerada pelo AG (testado anteriormente pelo Algoritmo Ótimo):

```
Qual algoritmo deseja executar?
1 - Algoritmo Exato (Força Bruta)
2 - Algoritmo Genético
2
Geração 0: Melhor custo = 437.0
Geração 4: Melhor custo = 421.0
Geração 912: Melhor custo = 410.0

Algoritmo Genético finalizado após 10000 gerações.
Melhor solução encontrada:
0 5 21 16 4 2 17 27 13 8 23 6 14 28 1 24 11 12 19 20 26 10 18 25 29 15 3 7 22 9
Custo total: 410.0
Tempo de execução: 7 segundos.
```

Para chegar ao resultado acima, foi utilizado os seguintes parâmetros:

```
private final int tamanhoDaPopulacao = 600; // alterar para calibrar o algoritmo (tamanho da população)
```

```
int maxGeracoes = 2000; // alterar para calibrar o algoritmo (quantidade de gerações)
double taxaDeMutacao = 0.1; // alterar para calibrar o algoritmo (0.05 = 5% de chance de mutação)
boolean elitismo = false; // alterar para calibrar o algoritmo (se true, o melhor indivíduo de cada geração é mantido na próxima geração)
int tipoMutacao = 3; // alterar para calibrar o algoritmo (1 = inserção, 2 = troca, 3 = inversão, 4 = mistura)
int tipoSelecao = 2; // alterar para calibrar o algoritmo (1 = roleta, 2 = torneio)
```

É importante ressaltar que alterar os parâmetros, pode diminuir o tempo de resposta do algoritmo, pois o código terá que fazer realizar mais tarefas, por exemplo, se aumentar o tamanho da população, aumenta o tamanho da matriz e pode acabar ficando mais lento o algoritmo. Aumentar o tamanho de gerações também causa esse efeito, porém, aumentar esses valores pode permitir chegar a uma resposta exata mais frequentemente. Vai depender da preferência do usuário, se o mesmo deseja uma resposta mais rápida, ou mais lenta, porém com grandes chances de obter a resposta exata.

Resultados Obtidos no Algoritmo Ótimo

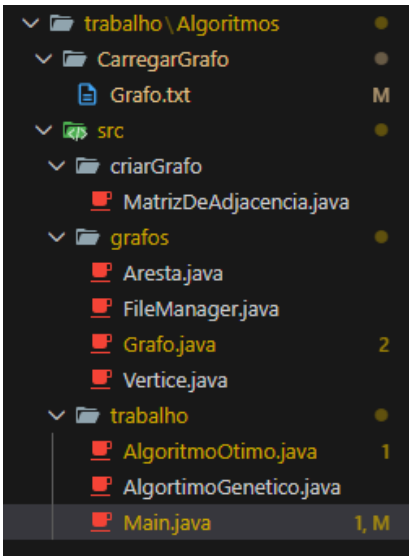
Com o uso da AG como poda, foi possível chegar ao resultado ótimo até mesmo em um caso no qual as cidades possuem pesos parecidos, podendo podar poucos galhos. Segue resultado:

```
Qual algoritmo deseja executar?
1 - Algoritmo Exato (Força Bruta)
2 - Algoritmo Genético
1
Geração 0: Melhor custo = 2603.0
Geração 469: Melhor custo = 2602.0

Algoritmo Genético finalizado após 5000 gerações.
Melhor solução encontrada:
0 21 16 15 4 1 22 8 17 3 11 10 25 9 20 6 2 24 19 5 13 23 14 18 7 12
Custo total: 2602.0
Limite Inicial (AG): 2602.0
Melhor Caminho (Kruskal Branch and Bound):
0 => 21 => 16 => 15 => 4 => 1 => 22 => 8 => 17 => 3 => 11 => 10 => 25 => 9 => 20 => 6 => 2 => 24 => 19 => 5 => 13 => 23 => 14 => 18 => 7 => 12 => 0
Melhor custo encontrado pelo Algoritmo Exato: 2602.0
Tempo de execução: 5 minutos.
```

Execução do código

Para executar o algoritmo, é preciso colocar o grafo em um arquivo chamado grafo.txt, na pasta “CarregarGrafo”:



Para executar o algoritmo genético, pode ser alterado os parâmetros do mesmo para verificar a solução com os diferentes tipos de soluções do AG.

```java
private final int tamanhoDaPopulacao = 600; // alterar para calibrar o algoritmo (tamanho da população)
```

```java
int maxGeracoes = 5000; // alterar para calibrar o algoritmo (quantidade de gerações)
double taxaDeMutacao = 0.15; // alterar para calibrar o algoritmo (0.05 = 5% de chance de mutação)
boolean elitismo = false; // alterar para calibrar o algoritmo (se true, o melhor indivíduo de cada geração é mantido na próxima geração)
int tipoMutacao = 3; // alterar para calibrar o algoritmo (1 = inserção, 2 = troca, 3 = inversão, 4 = mistura)
int tipoSelecao = 2; // alterar para calibrar o algoritmo (1 = roleta, 2 = torneio)
```

Conclusão

Esse trabalho demonstrou que é possível resolver instâncias de tamanho moderado do PCV (N = 26) em tempo hábil, desde que sejam aplicadas as técnicas rigorosas de podas e otimização de código. A utilização do Algoritmo Genético se provou ser uma alternativa muito boa para entradas muito grandes e provou que possui um valor muito grande atuando como limite superior ao algoritmo ótimo, mostrando que os dois combinados têm um potencial muito grande.