Trabalho Prático TSP

Pontifícia Universidade Católica de Minas Gerais

Guilherme Alberto de Moraes | 602206

Curso de Ciência da Computação, disciplina Projeto e Análise de Algoritmos

Professora Raquel Mini

Belo Horizonte, maio de 2017

Introdução

O Problema do Caixeiro Viajante (PCV, ou TSP do inglês Traveling Salesman Problem) é um problema que tenta determinar o melhor caminho a ser percorrido em um determinado grafo percorrendo cara um dos vértices apenas uma vez, e retornando a origem. Esse é um conhecido problema da área de Tecnologia da Informação devido à sua grande aplicabilidade e dificuldade em se trazer uma solução exata na medida em que o grafo cresce.

Esse trabalho traz como proposta a implementação de algoritmos de soluções do TSP na linguagem C++, sendo esses o Algoritmo Força Bruta, Algoritmo Branch and Bound e Algoritmo Genético. Os dois primeiros trazem soluções exatas para o TSP, enquanto o ultimo traz uma solução aproximada, possível de ser a solução ideal, mas sem essa garantia.

Implementação

Esse trabalho foi o primeiro implementado pelo autor na linguagem proposta. Por esse motivo, houve uma grande preocupação de se explorar ao máximo as funcionalidades da linguagem, havendo uma preocupação em tratamento de exceções, emprego de hierarquia na medida do possível, encapsulamento e legibilidade e reaproveitamento de código-fonte, permitindo a escalabilidade e simplificação do código, facilitando a sua leitura sem a necessidade de sempre recorrer a comentários e documentação para entender o princípio de funcionamento. Isso pode ser visto em todo o código, mas em especial na implementação da classe principal, a Source.cpp:

```
#include <iostream>
#include <vector>

#include "City.h"
#include "Solution.h"
#include "Util.h"
#include "BruteForceSolution.h"
#include "BranchAndBoundSolution.h"
#include "GeneticSolution.h"

using namespace std;

int main() {
	try {
		vector<City> cities = Util().loadCitiesFromFile();

		Solution bruteForceSolution = BruteForceSolution().loadSolution(cities);
		Util().generateFileResult("Brute Force Solution", bruteForceSolution);

		Solution brachAndBoundSolution = BranchAndBoundSolution().loadSolution(cities);
		Util().generateFileResult("Branch and Bound Solution", brachAndBoundSolution);
```

```
                Solution geneticSolution = GeneticSolution().loadSolution(cities);
                Util().generateFileResult("Genetic Solution", geneticSolution);

        }
        catch (runtime_error & e) {
                cerr << e.what() << endl;
        }
        return 0;
}
```

Podemos perceber que há a inclusão de cabeçalhos necessários para executar o essencial do problema, como o `Solution.h`, o qual é responsável por trazer uma solução dado um vetor de cidades, `BruteForceSolution.h`, o qual traz a solução de força bruta dado um vetor de cidades, etc. Para se obter maiores detalhes sobre cada um desses, é necessário apenas entrar no cabeçalho desejado. Por exemplo, uma solução é descrita como o seguinte:

```
#pragma once
#include <vector>
#include "City.h"

using namespace std;

class Solution
{
private:
        vector<City> cities;
        double distance;
public:
        Solution();
        Solution(vector<City>, double);
        ~Solution();

        vector<City> getCities();
        double getDistance();

        void setCities(vector<City> cities);
        void setDistance(double distance);
};
```

Ou seja, uma solução é encapsulada como um vetor de cidades e uma distância, que é exatamente o proposto como saída de uma execução na descrição deste trabalho prático.

Essa abordagem permite uma leitura facilitada do código fonte, sem a necessidade de recorrer somente à comentários no código para entender seu princípio de funcionamento, além de uma modelagem na qual as assinaturas dos métodos descrevem bem o seu comportamento de acordo com os parâmetros de entrada e saída.

Todo o código foi versionado no GitHub através do endereço https://github.com/guiAlberto/TSP, permitindo assim que a confecção de funcionalidades fosse feita de maneira escalável e controlada. Para não ocorrer problemas de plágio e exposição de informação antes da apresentação desse trabalho, esse repositório se encontra privado, porém se encontrará público uma vez que sua apresentação for realizada.

3

Relativo ao ambiente de desenvolvimento, o trabalho foi realizado em um PC Asus S451, com processador Intel Core i7-4500U @1.8GHz @2.4GHz, 8GB de RAM, com Windows 10 Home Single Language x64, versão 10.0.15063. Inicialmente foi utilizado o ambiente de desenvolvimento Eclipse Cpp Neon.3 Release (4.6.3) e compilador MinGW versão 5.3.0, porém, foram apresentadas limitações na IDE, especialmente na ferramenta de debug, fazendo com que o trabalho fosse mirado para o Microsoft Visual Studio Community 2017 versão 15.1 release 26403.7, com o compilador Microsoft C/C++ Optimizing Compiler Version 19.10.25019 for x64. Para executar o código fonte, simplesmente crie um projeto C++ vazio no Visual Studio, copie e cole os arquivos .cpp e .h na raiz do projeto (ou importe via git no repositório https://github.com/guiAlberto/TSP.git).

Quando a implementação de cada um dos métodos, temos o seguinte:

Algoritmo força bruta

Esse algoritmo trabalha com o arranjo das diferentes cidades a serem trabalhadas. No exemplo dado nesse trabalho, foi considerado um total de quatro cidades como um caso inicial (indicadas com valores numéricos 1, 2, 3 e 4). Sendo assim, o arranjo dessa cidade gera as seguintes possibilidades: 1234, 1243 ,1324 ,1342 ,1423, 1432, 2134, 2143, 2314, 2341, 2413, 2431, 3124, 3142, 3214, 3241, 3412, 3421, 4123, 4132, 4213, 4231, 4312, 4321. Como estamos tratando da permutação de n valores (sendo nesse caso, $n = 4$), teremos sempre um conjunto de $n!$ caminhos possíveis. Lembrando que para cada caso desses, temos que colocar ao final de cada elemento o primeiro, visto que estamos considerando a volta para o vértice inicial.

Algoritmo branch and bound

O algoritmo branch and bound trabalha com uma árvore de caminhos, as quais são incrementadas de acordo com o progresso de cada um dos vértices remanescentes, até se obter a melhor solução com o tamanho igual ao número de cidades mais 1 (que representa a volta para o vértice inicial). Como exemplo, considere o seguinte grafo:

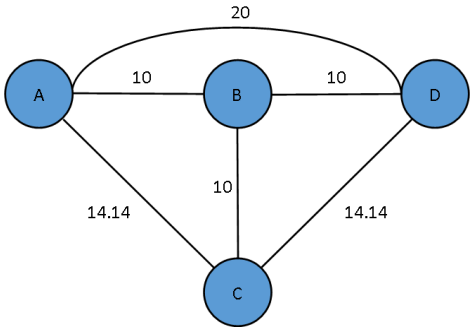

Figura 1 - Grafo exemplo para resolução do algoritmo branch and bound

Para resolver o TSP nesse problema usando o algoritmo branch and bound, assume um vértice qualquer no grafo como o estado inicial (vamos assumir o vértice A), o qual tem um custo zero para se chegar até lá. Ele ainda não é o caminho ideal, uma vez que o seu trajeto não possui cinco vértices. A partir desse momento, pode ser realizado um trajeto para B, C ou D. Nesse momento, ocorre a expansão dos caminhos filhos de A, como descrito na figura 2.

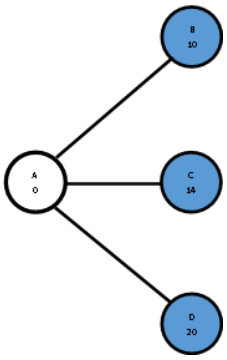

Figura 2 - Desenvolvimento do algoritmo branch and bound

Temos agora como possíveis soluções os caminhos AB, AC e AD, sendo AB o melhor deles até o momento. Logo, sobre ele, há a expansão para os vértices os quais ainda não foram percorridos, contendo os valores do vértice anterior mais o caminho do vértice final do caminho (B) até o próximo (C e D), não tendo mais a possível solução AB, mas sim ABC e ABD, com os valores 20 e 24.14 respectivamente.

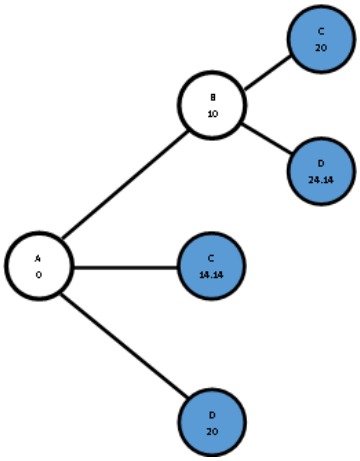

Figura 3 - Desenvolvimento do algoritmo branch and bound

Dessa forma, o processo de expansão do caminho de menor custo é repetido até que se tenha todas as arestas percorridas. Nesse momento, ocorre uma expansão extra que se trata da volta para o vértice original, trazendo assim um caminho com 5 vértices de acordo com esse exemplo. Quando tivermos o melhor caminho com um tamanho igual ao número de cidades mais um, temos então a solução.

Algoritmo genético

Esse algoritmo traz uma proposta interessante para a solução de um problema de otimização qualquer, porém sem a segurança do seu resultado ser exato. Para entender melhor seu funcionamento, precisamos definir alguns conceitos importantes para a sequência:

Cromossomo

Nesse contexto, um cromossomo representa uma solução qualquer para um problema, seja ela ótima ou não. Podemos entender que um cromossomo é um indivíduo qualquer, e não necessariamente que um indivíduo é uma composição de uma grande quantidade de cromossomos para formar sua unidade.

População

Aqui, uma população é um conjunto finito de indivíduos. Como estamos assumindo que um cromossomo é um indivíduo, uma população é então um conjunto de cromossomos.

Mutação

Uma mutação é uma alteração no cromossomo dado um estado inicial. Sendo assim, um cromossomo pode passar por mutações ao ter sua estrutura base alterada de acordo com algum critério.

Reprodução

Uma reprodução é a geração de um novo indivíduo (ou seja, um novo cromossomo) dada a conjunção de dois outros.

Seleção Natural

A seleção natural é o processo de eliminação de indivíduos em uma população de acordo com uma função objetivo. Indivíduos mais propícios a sobreviver no ambiente (ou seja, mais próximos da função objetivo) sobrevivem, enquanto os mais distantes são eliminados.

Geração

A geração é a população resultado dado o conjunto de processos de mutação, reprodução e seleção natural.

Visto isso, podemos entender que um cromossomo no TSP é um caminho qualquer que parte de uma origem, retorna a origem e passa por todos os vértices apenas uma vez. Uma população é um conjunto de soluções do TSP. Uma mutação é uma alteração na composição base da solução, ou seja, é uma mudança no trajeto do TSP. Uma reprodução é uma composição parcial de dois cromossomos, nesse caso, uma composição parcial dos trajetos. Por fim, uma seleção natural é um filtro sobre uma população a qual aproveita os cromossomos que mais se aproxima da função objetivo (obtenção do menor custo em um caminho).

Essa solução se mostra interessante uma vez que, dado uma população bem distribuída no espaço de soluções, as mutações e reproduções trazem maiores possíveis soluções de acordo com a população atual, e a seleção natural filtra as melhores soluções. Tendo então um número limitado de gerações, o resultado tende a se aproximar do ideal, porém sem nunca ter o critério exato de parada, uma vez que uma solução por si só não tem a capacidade de saber se é a ótima.

Análise de complexidade

Para cada uma das soluções propostas, existe um método que simplesmente traz a resposta de acordo com a entrada (como pode ser visto no código-fonte de `Source.cpp`, há três invocações do método `Solution solution = SolutionType().loadSolution(cities)`), sendo o primeiro a solução através do algoritmo de força bruta, o segundo através do algoritmo branch and bound e o terceiro através do algoritmo genético. Sendo assim, a operação relevante para cada uma das abordagens é a `.loadSolution()`. Assim, a complexidade em seu maior nível de abstração para todos os algoritmos pode ser obtida analisando somente o processamento desses três métodos de acordo com o tamanho da entrada n (quantidade de cidades).

O algoritmo força bruta trabalha com o arranjo das cidades fornecidas. Sendo assim, ele irá trazer $n!$ caminhos dado uma entrada de n cidades, irá percorrer esses caminhos um a um até encontrar o menor deles. Dessa forma, seu tempo de execução se dá em $\Theta(n!)$.

O algoritmo branch and bound explora a árvore de possibilidades, trazendo sub caminhos da solução até chegar ao critério de parada (esse sendo a melhor solução com o tamanho mínimo que possa ser considerado como tal). Dessa forma, o melhor caso possível é encontrar sequencialmente a expansão dos nodos filhos da árvore que logo trazem a solução. Porém o pior caso é quando se torna necessário expandir todos os nodos da árvore até chegar na última folha, sendo essa a solução, sendo então igual à solução de força bruta. Devemos lembrar também que seu funcionamento exige que o trajeto de volta para o vértice de origem deve ser considerado. Sendo assim podemos afirmar que sua solução dá através das funções $O(n!)$ e $\Omega(n)$.

Por fim, o algoritmo genético traz como proposta uma solução polinomial de acordo com a entrada. Da forma como ele foi implementado, dado um conjunto de entrada n, será gerado uma população inicial distribuída com n cromossomos, e um total de 5 gerações. Dessa forma, a solução desse algoritmo se encontra em $\Theta(n)$.

Testes

O cenário de teste montado consiste em 10 documentos de textos, cada um deles trazendo uma quantidade entre 1 e 10 vértices em um grafo com valores gerados aleatoriamente. Inicialmente foi colocado os algoritmos para trabalharem em sequência, porém a partir do grafo com 8 vértices, não foi apresentado solução em mais de uma hora de espera. Esse teste gerou o seguinte resultado:

Número de vértices	Algoritmo Força Bruta	Algoritmo Branch and Bound	Algoritmo Genético
1	0	0	0.01
2	0	0.002	0.012
3	0.001	0.003	0.012
4	0.011	0.033	0.07
5	0.232	0.192	0.022
6	5.087	1.784	0.027
7	245.13	28.016	0.049

Tabela 1 - Resultado geral do primeiro teste

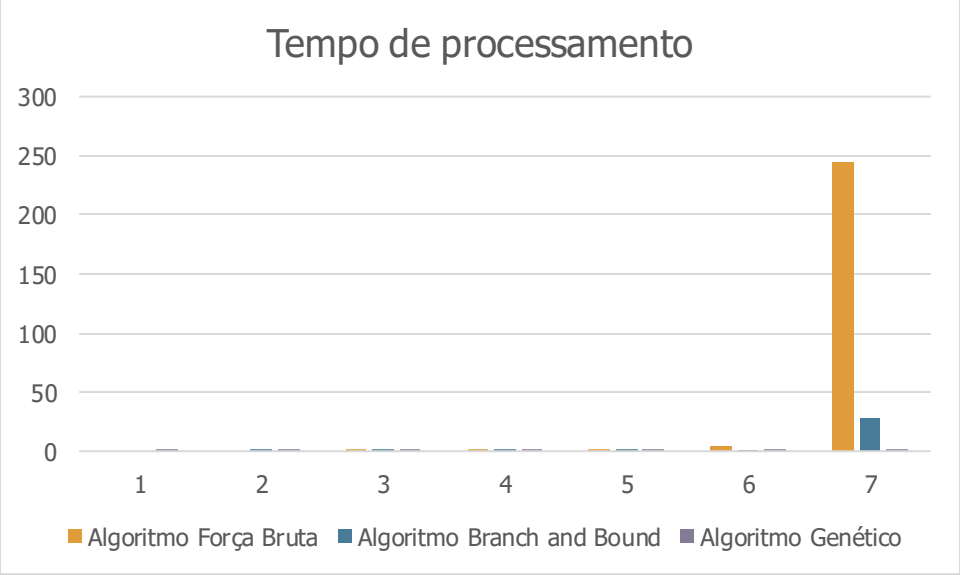

Gráfico 1 - Resultado geral do primeiro teste

Como a própria visualização dos algoritmos ficou comprometida com o resultado, além de não trazer os valores de análise para todas as hipóteses, foi mudada a abordagem de forma que cada um dos algoritmos fosse processado individualmente. Assim, foi gerado novos valores para cada um dos algoritmos.

O algoritmo força bruta novamente não entregou o resultado para 8 vértices após uma hora de espera. Foi gerado então os seguintes dados:

Número de vértices	Algoritmo Força Bruta
1	0
2	0
3	0.001
4	0.007
5	0.211
6	5.28
7	253.967

Tabela 2- Tempo de execução do algoritmo força bruta

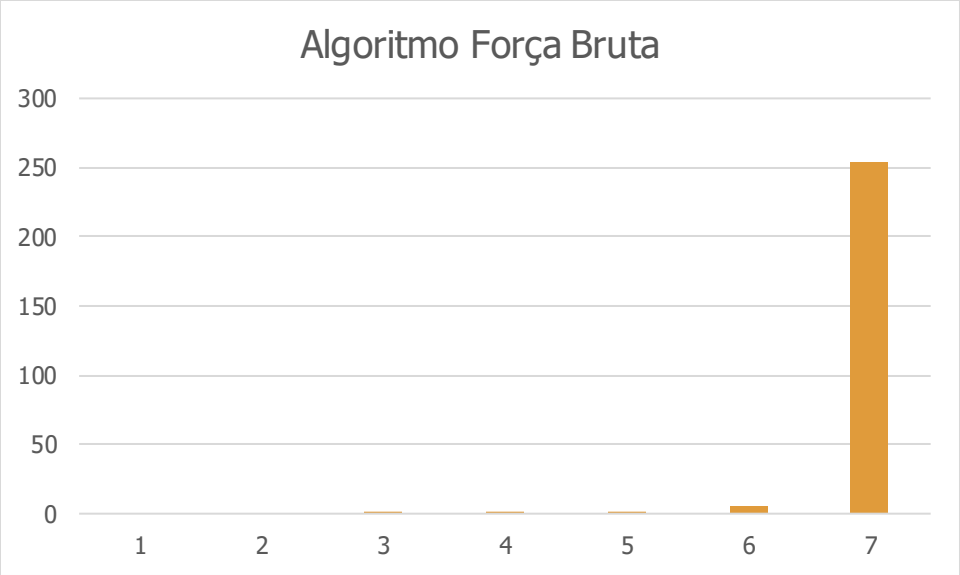

Gráfico 2- Tempo de execução do algoritmo força bruta

Já o algoritmo branch and bound conseguiu trazer um resultado para 8 vértices, porém novamente após uma hora de espera, não houve resposta.

Número de vértices	Algoritmo Branch and Bound
1	0
2	0.002
3	0.003
4	0.033
5	0.192
6	1.784

Número de vértices	Algoritmo Branch and Bound
7	28.016
8	721.398

Tabela 3- Tempo de execução do algoritmo branch and bound

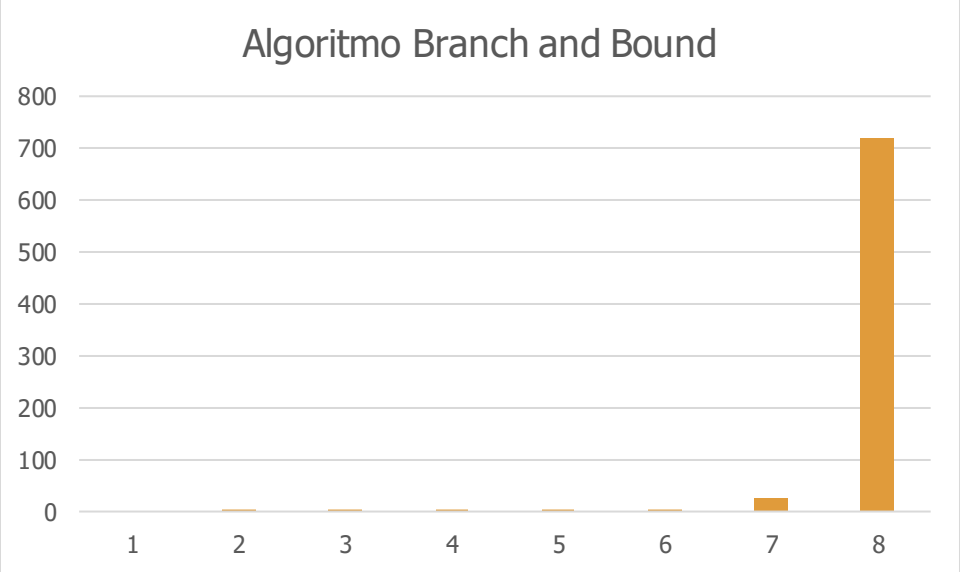

Gráfico 3- Tempo de execução do algoritmo branch and bound

Por fim, o algoritmo genético conseguiu facilmente percorrer todas as possibilidades, gerando os resultados:

Número de vértices	Algoritmo Genético
1	0.012
2	0.019
3	0.013
4	0.018
5	0.049
6	0.043
7	0.058
8	0.07
9	0.063
10	0.1

Tabela 4- Tempo de execução do algoritmo genético

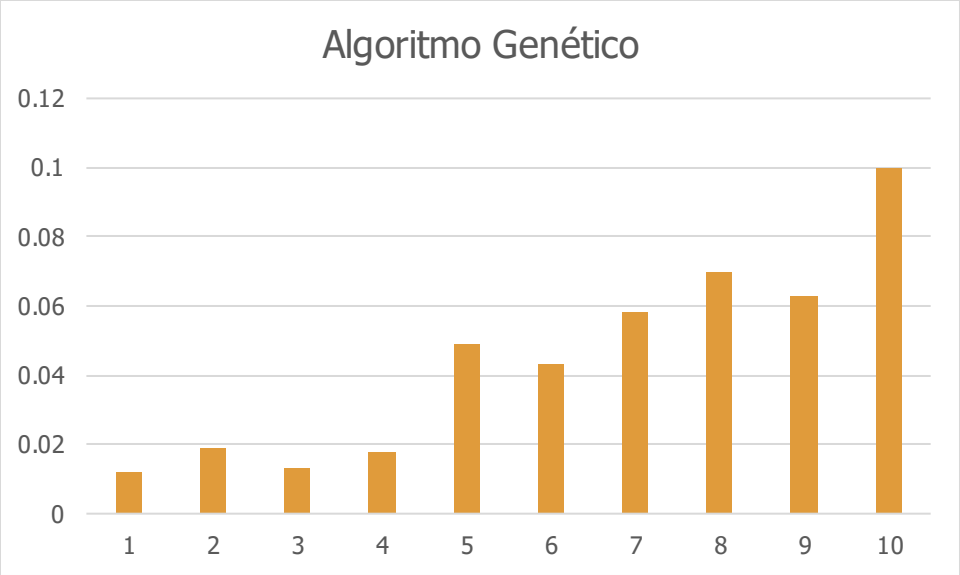

Gráfico 4- Tempo de execução do algoritmo genético

Vale destacar também que o algoritmo força bruta e genético encontraram para $n = 7$ um caminho ótimo com valor de 2126.72, enquanto o algoritmo genético encontrou um valor de 2910.31, aproximadamente 37% a mais do caminho ótimo.

Conclusão

Nesse trabalho foi desenvolvido três maneiras diferentes de resolver o problema do caixeiro viajante. Durante a fase de testes, foi notório a dificuldade de se encontrar uma solução ótima, mesmo com um número relativamente pequeno de vértices dentro do grafo utilizando algoritmos exatos. Mais marcante do que isso, a abordagem proposta com o algoritmo genético se mostrou incrivelmente eficiente mesmo utilizando heurísticas bastante simples. O estudo e implementação de heurísticas para melhor tratar os métodos de mutação, reprodução e geração certamente podem trazer a esse algoritmo uma acurácia ainda maior do que a que foi encontrada nesse resultado.

Bibliografia

- LAPORTE, Gilbert. The traveling salesman problem: An overview of exact and approximate algorithms. European Journal Of Operational Research, [s.l.], v. 59, n. 2, p.231-247, jun. 1992. Elsevier BV. http://dx.doi.org/10.1016/0377-2217(92)90138-y.

- MÜHLENBEIN, H.. Parallel genetic algorithms, population genetics and combinatorial optimization. Parallelism, Learning, Evolution, [s.l.], p.398-406, 1991. Springer Berlin Heidelberg. http://dx.doi.org/10.1007/3-540-55027-5_23.
- BRAUN, Heinrich. On solving travelling salesman problems by genetic algorithms. Parallel Problem Solving From Nature, [s.l.], p.129-133, 2005. Springer-Verlag. http://dx.doi.org/10.1007/bfb0029743.