Trabalho de Implementação 02

Matéria: Heurísticas e Metaheurísticas **Professor**: Thiago Ferreira de Noronha
Aluno: Guilherme Teres Nunes **Matrícula**: 2016077187

Problema do Caixeiro Viajante

Esse trabalho consiste em apresentar uma heurística baseada em VND para resolver o problema clássico do Caixeiro Viajante, ou **Traveling Salesman Problem (TSP)**. Esse problema consiste em: dado uma lista de cidades “E” e suas coordenadas (ou distâncias entre elas), calcular qual a menor rota que o caixeiro viajante pode percorrer, começando de uma cidade X e terminando no mesmo local, passando apenas uma vez em cada cidade, para visitar **todas** as cidades da lista “E”.

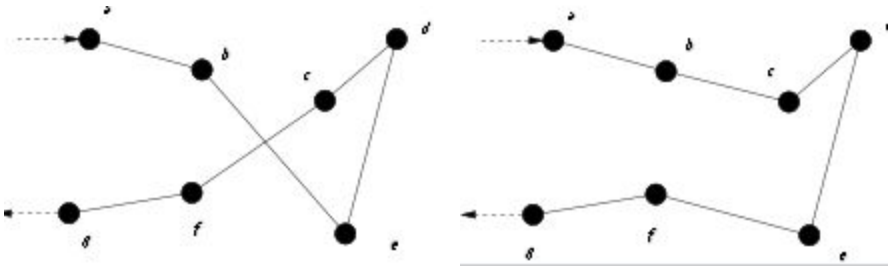
Esse é um problema NP-Difícil extremamente conhecido e popular no meio da computação, exatamente por sua aplicabilidade prática e pela vasta opção de heurísticas para resolvê-lo.

Heurística: Nearest Neighbor + 2-OPT

A partir da solução e implementação feita no trabalho prático 01 (Heurística Nearest Neighbor para TSP), foi implementado o algoritmo de busca local 2-OPT, que primeiro calcula um caminho inicial usando a primeira parte do trabalho.

O 2-opt é um algoritmo de busca local para resolver o problema do Caixeiro Viajante, onde a principal ideia é identificar onde a rota do viajante cruza com ela mesmo e reordená-la de forma que isso não aconteça.

A imagem a seguir, por exemplo, mostra a rota antes e depois de executar o 2-opt:



Fonte: https://en.wikipedia.org/wiki/2-opt

Uma abordagem interessante (e que foi implementada) é testar todas as trocas de rotas possíveis e verificar se houve uma diminuição do percurso do viajante. Se sim, é uma alternativa melhor.

Implementação

A linguagem de programação utilizada foi o C++14. O código também está disponível no github através desse link:

https://github.com/UnidayStudio/UFMG-TSP-Heuristics

O algoritmo funciona da seguinte forma:

Primeiro ele gera uma rota inicial, utilizando o Nearest Neighbor implementado no trabalho prático anterior. Depois, ele executa A vezes, onde A é a quantidade máxima de iterações do algoritmo, passado como parâmetro para a função, o seguinte procedimento:

considere C como a quantidade de cidades percorridas na rota.
***para cada** i, onde i = [1, … C - 3[, executar:*
    ***para cada** k, onde k = [i + 1, … C - 2[, executar:*
        nova rota = rota original performando o swap de i até k.

***se** a distância da nova rota for menor que a rota original:*
considerar a nova rota como a original e
***finalizar** o procedimento*

Perceba que o procedimento só contabiliza uma troca bem sucedida (quando ele finaliza após observar uma redução do caminho). Por isso foi implementado o sistema de iterações, para executar esse procedimento múltiplas vezes até obter um resultado satisfatório.

Também seria possível executá-lo até que nenhuma melhoria na rota fosse observada. Para fazer isso, basta passar um número de iterações extremamente alto para a função. Ela possui uma condição de parada para quando o procedimento acima for executado por completo sem fazer nenhuma alteração (significando que ele já encontrou sua melhor solução possível, já que nenhum caminho alternativo foi mais eficiente). Esse detalhe de implementação foi omitido do pseudo código acima para simplificar a explicação do procedimento.

A troca da rota (swap), chamada de “Swap 2-OPT” é feita da seguinte forma: Ela recebe o caminho e dois pontos, de início e fim do trecho a ser reordenado. Os trechos da rota do início até o primeiro ponto e do segundo até o fim são mantidos da mesma forma. Mas o trecho entre eles é invertido, performando o “swap”.

Resultados

Cada uma das entradas e seus respectivos algoritmos de distâncias foram executadas **mil vezes** para obter a média de tempo de execução. Para fins de referência, o custo obtido executando apenas o Nearest Neighbor foi inserido na tabela.

Abaixo a tabela com os resultados obtidos:

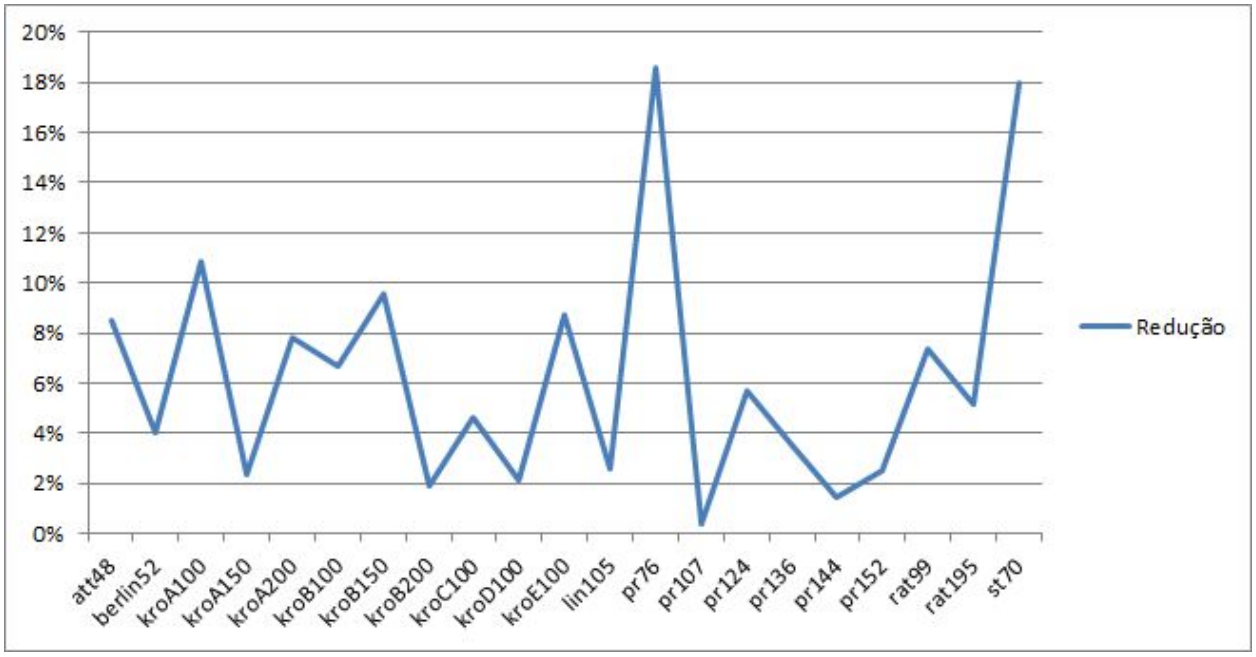
Valores de Referência		**Trabalho**			
Instância	Solução ótima	Custo NN	Custo NN + 2-OPT	Tempo (s) NN + 2-OPT	gap (%)
att48	10628	12813	11727	0.016	10.34%
berlin52	7542	8980	8621	0.006	14.31%
kroA100	21282	27807	24787	0.027	16.47%
kroA150	26524	33633	32840	0.068	23.81%
kroA200	29368	35859	33043	0.104	12.51%
kroB100	22141	29158	27200	0.206	22.85%
kroB150	26130	34499	31194	0.096	19.38%
kroB200	29437	36980	36261	0.441	23.18%
kroC100	20749	26227	25001	0.031	20.49%
kroD100	21294	26947	26370	0.032	23.84%
kroE100	22068	27460	25061	0.080	13.56%
lin105	14379	20356	19822	0.027	37.85%
pr76	108159	153462	124987	0.096	15.56%
pr107	44303	46680	46482	0.286	4.92%
pr124	59030	69297	65323	0.260	10.66%
pr136	96772	120769	116533	0.046	20.42%
pr144	58537	61652	60751	0.033	3.78%

pr152	73682	85699	83550	0.057	13.39%
rat99	1211	1554	1440	0.183	18.91%
rat195	2323	2752	2609	0.691	12.31%
st70	675	830	681	0.039	0.89%
				Média	16.16%

A partir dos resultados gerados na primeira etapa do trabalho e dos atuais, foi possível calcular a porcentagem de redução do custo do resultado obtido pelo Nearest Neighbor através do 2-OPT. Seguindo a fórmula:

Redução = 1.f - custo final do NN e 2-OPT / custo inicial do NN

Abaixo os resultados obtidos (mais é melhor):



Conclusão

Houve uma redução significativa na distância percorrida pelo caixeiro viajante ao executar o 2-OPT. Em alguns casos, como nas rotas pr76 e st70, gerando um caminho final quase 20% menor que o original. O gap entre os resultados obtidos e a solução ótima foi de cerca de 16%, consideravelmente menor que os resultados obtidos ao executar apenas o Nearest Neighbor (no trabalho prático 01), onde o gap chegou próximo de 25%.

Entretanto, isso também significou um aumento considerável no tempo de execução. A média de tempo de execução da primeira parte do trabalho foi de 0.000319 segundos. Nesta segunda parte, de 0.1345 segundos. Houve um aumento de mais de quatrocentas vezes no tempo.

Ainda sim, é um valor consideravelmente baixo para várias aplicações como o cálculo de rotas em um aplicativo de mapas. Para outras, como na inteligência artificial de um agente em um jogo digital que precisa ser executado em tempo real, com um tempo de execução máximo de um frame de 16ms (para que o jogo possa ser executado a 60 quadros por segundo), não. A não ser que esse cálculo seja executado de forma assíncrona e o jogo não dependa de seu resultado imediato.

Referências

- **Local Search in Combinatorial Optimization**. Arts and Lenstra J. K., John Wiley and Suns, 1997. (Capítulo 8)