O Problema do Caixeiro-Viajante - Parte 3

Aluno: João Vitor Venceslau Coelho

Executando a Metaheurística

Executando a metaheurística VNS (*Variable Neighborhood Search*) implementada na instância ***ulysses22*** utilizando os seguintes parâmetros:

- Maior Vizinhança a considerar: **4**
- Número máximo de iterações: **500**
- Número máximo de iterações sem melhora: **250**
- Tempo máximo de execução: **30 minutos**
- Melhor solução inicial: **gerada por algoritmo guloso**

Além do seguinte parâmetro para reprodutibilidade:

- Seed de geração de números pseudo-aleatórios: **42**

E os seguintes critérios de parada:

- Número de iterações alcançou o máximo estabelecido;
- Número de iterações sem melhora alcançou o máximo estabelecido;
- Tempo de execução do algoritmo alcançou o máximo estabelecido;

Foi obtido o seguinte resultado:

- Tour: **[17, 22, 4, 18, 8, 1, 16, 13, 7, 6, 5, 15, 14, 12, 21, 20, 19, 11, 9, 10, 3, 2]**
- Com custo: **7953**
- Após executar por **02** minutos e **15** segundos
- Após **279** iterações
- E última melhora na iteração **28**

Algoritmo Escolhido (VNS)

O algoritmo *Variable Neighborhood Seach* (VNS) implementado é uma mistura das variantes do VNS apresentadas em aula com a adoção de algumas das ideias apresentadas durante o curso. Abaixo está o pseudocódigo do algoritmo junto a uma breve explicação.

```
VNS(k_max, iter_max, sem_melhora, tempo_max, melhor_solucao):
 se melhor_solucao igual a NULL faça:
  melhor_solucao ← gerar_tour_aleatorio
 solucao ← copiar melhor_solucao
 k ← 1
 enquanto (passo atual menor que o máximo e número de passos sem melhora menor que o máximo e tempo de execução do algoritmo menor que o máximo) faça:
  se (k maior que k_max) faça:
   marcar solucao atual como explorada
   solucao ← gerar_tour_aleatorio
   se solucao já foi explorada faça:
    volte para o começo do laço
   k ← 1
  solucao ← encontrar_primeiro_melhor_vizinho(solucao, k)
  se custo(solucao) melhor que custo(melhor_solucao) faça:
   melhor_solucao ← solucao
   k ← 1
```

```
   senão faça:
     k ← k + 1
retorne melhor_solucao
encontrar_primeiro_melhor_vizinho(solucao, k):
  melhor_solucao ← copiar solucao
  vizinhanca ← gerar_vizinhaca(solucao, k)
  para cada vizinho na vizinhaca, faça:
    se custo(vizinho) melhor que custo(melhor_solucao) faça:
      retorne vizinho
retorne melhor_solucao
```

O algoritmo implementado inicia com uma solução aleatória, caso não seja fornecida uma solução inicial, e busca nos seus vizinhos alguma solução melhor, retornando a primeira que for encontrada. Os vizinhos considerados estão numa vizinhança definida por **k**, sendo que quando **k** é 1, os vizinhos são as soluções geradas a partir da original, alterando apenas um par de cidades adjacentes na solução, após alterar todos os pares, o conjunto de soluções geradas é a vizinhança **k=1** da solução. Para gerar a vizinhança **k=2**, considera-se a vizinhança **k=1** de todos os vizinhos existentes na vizinhança **k=1** da solução original, além da própria vizinhança **k=1**. As vizinhanças **k=n** seguem o mesmo padrão, considerando a vizinhança **k=(n-1)** dos vizinhos encontrados na vizinhança **k=1**.
Após encontrar o melhor vizinho da solução, se esse melhor vizinho for melhor que a melhor solução conhecida, atualizamos a melhor solução e utilizamos ela na próxima busca, senão, aumentamos o **k** que define a vizinhança a ser considerada na busca.
Se o **k** atingir o máximo definido, geramos uma nova solução aleatória e buscamos seu melhor vizinho, em vez de continuar usando a solução anterior. Caso essa solução aleatória já tenha sido explorada, voltamos ao início do laço e geramos outra.
As soluções são representadas por um vetor de números, onde cada número representa uma cidade a ser visitada, no *Python* esse vetor é uma lista de inteiros. Essa lista é encapsulada na classe *Solve* que implementa algumas funções auxiliares, além de armazenar a lista, seu tamanho e o custo da solução.

Motivação para as mudanças

A geração de novas soluções aleatórias em vez de interromper a execução tem como finalidade aumentar a diversidade de soluções exploradas, evitando parar em um mínimo local.
A adição do tempo como condição de parada é para evitar que o algoritmo fique muito tempo executando, pois apenas considerando o número de iterações total e o número de iterações sem melhora, muito tempo ainda era gasto na execução do algoritmo.
O conjunto de soluções exploradas tem como finalidade evitar gastar processamento em soluções que já foram exploradas, tanto na busca dentro da vizinhança como na geração de novas soluções.

(EXTRA) Comparação GRASP - VNS

Antes de implementar o VNS, implementei o GRASP (*Greedy Randomized Adaptative Search Procedure*) mostrado em aula, e decidi comparar a execução dos dois. Algumas modificações foram feitas no GRASP também, porém não serão descritas em detalhe, sendo elas: a adição do **tempo como critério de parada** e o **incremento do *alpha*** após um certo número de iterações sem melhora, além é claro da função de **criação da solução**, pois o problema abordado é diferente do mostrado em aula.

Os parâmetros utilizados no GRASP foram:

- *alpha* inicial: **0.0**
- Tamanho da vizinhança: **3**
- Iterações máximas: **1000**
- Máximo de Iter. sem melhora: **500**
- Iter. antes do incremento do *alpha*: **10**
- Tempo máximo: **30 minuto**

Segue a tabela com a execução dos dois algoritmos em 17 das 30 instâncias pesquisadas:

Nome	Tamanho	Melhor	Greed	Custo VNS	Tempo VNS	Custo GRASP	Tempo GRASP	Melhor
burma14	14		4048	3323	18s	3490	50s	VNS
ulysses16	16	6859	9988	6909	32s	6909	1m 21s	VNS
gr17	17		2187	2085	40s	2085	1m 37s	VNS
gr21	21		3333	3303	1m 37s	2933	3m 12s	GRASP
ulysses22	22	7013	10586	7953	2m 15s	7781	3m 49s	GRASP
gr24	24	1276	1553	1332	2m 59s	1355	4m 52s	VNS
fri26	26	937	1112	1034	3m 19s	955	7m 05s	GRASP
bayg29	29	1610	2005	1838	5m 44s	1902	9m 45s	VNS
bays29	29	2020	2258	2111	5m 20s	2125	9m 50s	VNS
dantzig42	42		956	857	30m	846	30m	GRASP
swiss42	42		1630	1418	30m	1376	30m	GRASP
att48	48	10628	12861	12218	30m	12572	30m	VNS
gr48	48	5046	6098	5780	30m	5851	30m	VNS
hk48	48		13181	12696	30m	12181	30m	GRASP
eil51	51	426	511	505	30m	475	30m	GRASP
berlin52	52	7542	8980	8279	30m	8813	30m	VNS
brazil58	58	25395	30774	29288	30m	26961	30m	GRASP

* Os espaços vazios na coluna **Melhor** indicam que não possui um valor é conhecido

Das 17 instâncias utilizadas, o VNS obteve um melhor resultado em 9 delas enquanto o GRASP foi melhor nas 8 restantes, sendo que em 8 das 17 instâncias, tanto o VNS quanto o GRASP tiveram sua execução interrompida por alcançarem o tempo máximo de execução, sendo essas as instâncias de maior tamanho, possuindo mais de 40 nós.