Soluções para problemas difíceis

Filipe Rodrigues Batista de Oliveira[1]

[1]Universidade Federal de Minas Gerais (UFMG)

frbo@ufmg.br

1. Introdução

O problema do caixeiro viajante euclidiano é um problema de otimização referente a grafos ponderados onde os pesos das arestas são métricas, ele é NP-completo e tem aplicações em várias áreas; A definição do mesmo consiste em encontrar um circuito hamiltoniano com menor peso. Para o presente trabalho foi implementado um algoritmo branch-and-bound e uma heurística associada (para estimar o limiar de custo de se percorrer um determinado ramo da árvore de estados), além dos algoritmos aproximativos: ”twice-around-the-tree”e algoritmo de Christofides, em seguida faremos uma discussão sobre os resultados obtidos em dez de setenta e nove instâncias de [**?**], segundo as seguintes métricas: qualidade da solução, tempo e espaço.

2. Implementação

Todo o trabalho foi concebido na linguagem Python3.8 e as bibliotecas (além da padrão) usadas foram Pandas e Networkx para entrada e saída dos dados, operações básicas com grafos. A máquina utilizada tanto para desenvolvimento quanto para teste foi uma com: Intel Core i5-2450M 2.50 GHz; 8 Gb de memória RAM DDR3 e o sistema operacional instalado na mesma é um GNU/Linux (Debian 11).

Antes de começar de fato a implementação, foram escritos alguns casos de teste afim de verificar a corretude dos algoritmos que seriam implementados, visando também economizar tempo de desenvolvimento, já que para grafos com um grande número de nós o algoritmo de branch-and-bound pode demorar bastante, e re-executar o programa incorreria em grande desperdício de tempo.

Procurou-se seguir boas práticas de modularização e as implementações dos algoritmos estão divididas em três arquivos: main.py, bnb.py, aprox.py.

No último arquivo estão as implementações dos algoritmos aproximativos ”twice-around-the-tree”e o algoritmo de Christofides, e os mesmos são traduções para Python dos pseudo-códigos em [Vimeiro], como a linguagem escolhida é bem expressiva, a tradução foi bastante natural.

No arquivo bnb.py está a implementação do algoritmo branch-and-bound e da heurística para estimar o limiar superior de custo de se percorrer um determinado ramo da árvore de estados, sendo que a ideia da última é: encontre as duas arestas de menor peso que incidem em cada vértice e some os pesos das duas, repetindo até que terminamos de percorrer todos os nós do grafo, então nós somamos todos os valores obtidos para cada nó (seguindo a descrição precedente) e dividimos o valor total por dois; Para realizar isso, nós obtemos os índices que se referem as colunas com os menores valores da linha da matriz de adjacência do grafo correspondente ao nó corrente (retirando o primeiro, que sempre corresponde a diagonal principal e sempre assume o menor valor: 0, já que o grafo

não tem laços), depois nós verificamos se o nó corrente já está no caminho (postulante a circuito hamiltoniano), se sim, nós temos que obrigatoriamente incluí-lo, independente se um dos dois menores valores está em outro nó. E necessário dizer que racionalizar e por essa ideia na forma de um algoritmo foi a etapa mais desafiadora, essa dificuldade deve-se as operações adicionais que devem ser realizadas no caso em que o nó já está no caminho.

A implementação do algoritmo de branch-and-bound foi basicamente uma tradução do pseudocódigo em [Vimeiro] para a linguagem Python, a única tarefa original (em forma) foi a subrotina descrita no parágrafo anterior; O mesmo usa a estratégia de ”best-first”para explorar os ramos da árvore de estados, sendo implementada através de uma fila de prioridades (usando um Heap).

Para o primeiro arquivo: main.py, nós baixamos as bases de dados usadas a serem utilizadas e filtramos os dados (coordenadas x e y dos nós), dando origem a um Dataframe, em seguida nós transferimos esses dados para um Grafo completo (implementado pela biblioteca networkx), em seguida, nós definimos os pesos das arestas (já que inicialmente elas não têm peso) criando um atributo ”weight”(que é o padrão na biblioteca para grafos ponderados) e atribuimos como valor para este atributo os resultados dos cálculos das distâncias euclidianas de um nó a outro. Dessa forma, nós construímos os grafos de todas as bases de dados, armazenando todas elas em uma lista. Por conta da incerteza a respeito do tempo de execução do algoritmo de Branch-and-bound (já que o algoritmo é exato e no pior caso tem que visitar todas os nós do grafo) nós limitamos o tempo de execução à 30 minutos, ou seja, todos as execuções que atingissem este valor eram abortadas e os dados referentes à execução (tempo gasto, espaço gasto, qualidade da solução) eram colocados como NA (não disponível) em seus respectivos dicionários (de tempo, espaço, ...), finalmente, convertemos esses para um novo Dataframe e os retornamos em um formato .csv.

3. Análise Experimental

Para executar o programa abra um terminal e digite:

```
$ python3 main.py [N]
```

onde:

- **[N]** := Os N primeiros datasets serão baixados e executados pelo programa; Se não especificado, todos os datasets no arquivo ”tp2 datasets”serão executados;

Obs: É necessário conexão com a internet, para que o programa acesse as bases de dados.

Cada base de dados levou (no mínimo) 30 minutos para ser processada, como foram coletados dados de execução para dez bases de dados, foram necessário cerca de cinco horas para que o programa terminasse de executar.

Infelizmente, não foi possível completar os experimentos para o trabalho a tempo de o prazo de entrega acabar, devido ao grande tempo de execução do algoritmo de Branch-and-Bound para instâncias grandes. Mas os dados estão disponíveis no repositório deste trabalho e é de se esperar que os algoritmos branch-and-bound tenham disparado o maior tempo, embora com melhor qualidade de solução, enquanto os algoritmos de Christofides e twice-around-the-tree tenham menor tempo e qualidade de solução razoável, embora não tão bons quanto branch-and-bound, então no caso onde queremos uma boa solução rapidamente é melhor optar pelos algoritmos aproximativos, especialmente o de Christofides que apresenta solução mais próxima do ótimo.

4. Conclusão

Com os experimentos acima nós podemos ver quão útil são os algoritmos aproximativos, eles nos fornecem valores razoáveis (embora nunca encontrem o ótimo!) em detrimento a um tempo de execução muito mais rápido.

Referências

Vimeiro, R. *Slides da disciplina DCC207*.