U Positivo - UTFPR - PUCPr – 11/02/2019 - 12:35:38.0
Prof Dr P Kantek (pkantek@gmail.com)
Estratégias de Programaçaão 2 VIVO666c, V: 1.05 1
- 1 _____ / _____ / _____

Esta folha está baseada no excelente livro BHARGAVA, Aditya. **Entendendo Algoritmos** - um guia ilustrado para programadores e outros curiosos. São Paulo: Novatec, 2017.

Tabelas Hash

Uma busca linear demora $O(n)$, uma busca binária demora $O(log_2\ n)$. Que tal seria ter uma busca que demorasse $O(1)$? Bem melhor, não é ? Esta busca existe e é feita usando-se uma tabela hash. Uma tabela hash usa como índice o resultado de uma função hash. Esta função recebe um valor (o item a ser procurado) e devolve o seu índice na lista original. Tal função deve ter 2 requisitos:

- ser consistente: cada vez que o mesmo item for consultado ela deve devolver o mesmo valor
- itens diferentes devem gerar índices diferentes

Como o espaço de chaves (itens) sempre é muito maior do que o espaço de índices, deve-se pensar no que fazer quando ocorrer a **colisão**. Trata-se do que acontece quando dois itens diferentes são mapeados para o mesmo índice. Na linguagem Python, tabelas hash são chamadas de dicionários, e usando-os você nunca terá que se preocupar com as colisões: é problema do Python. Python reconhece dicionários pelo uso de chaves {}. Acompanhe o exemplo: seja criar uma lista de preços de itens de papelaria

```
precos = {}
precos["folha"]=0.03
precos["lapis_b2"]=2.20
precos["grampeador"]=18.90
precos["borracha"]=5.50

print(precos)
```

A consulta a um item é feita fazendo `>>> precos["folha"]` e ele responde 0.03. Já a consulta a um item inexistente como `>>> precos["cartolina"]` gera um `KeyError`. Se quiser testar a existência antes de deixar acontecer o erro use

```
if precos.get("cartolina"):
  print("existe este item")
else:
  print("nao existe este item")
```

Não vai se explicar aqui como gerar a função hash, já que o Python (e a maioria das linguagens modernas) faz isso para você. Se quiser, busque este tópico em qualquer livro de estruturas de dados. Ou no `VIVO521`. É poesia (matemática) pura.

Busca em Largura

Grafos

Um grafo é um objeto matemático composto por pontos (vértices) e conexões (arestas). Qualquer mapa pode ser entendido como um grafo, senão vejamos:

tipo	vértice	aresta
mapa rodoviário	cidade	estrada
rede computadores	computador	link
amigos do FB	usuário	amizade
curso de engenharia	disciplinas	pré-requisitos
família	pessoas	ascendência
fluxograma	atividades	precedências

Um vértice pode estar isolado, conectado a um outro vértice, a muitos vértices ou a todos os vértices. A conexão pode ser unidirecional (de a para b) ou bidirecional. Neste último caso, a bidirecionalidade pode ser entendida como havendo os dois sentidos. Ou seja a conexão de a e b sem sentido, pode ser vista como tendo os dois sentidos (de a para b e de b para a).

Nosso primeiro algoritmo de grafos é saindo de um vértice, percorrer todos os vértices do grafo. Se houver vértices isolados, este algoritmo não tem solução. Se cada vértice só estiver conectado a outros 2 pode haver uma solução trivial. Mas nossa tarefa é resolver o caso mais geral: cada vértice tem muitas conexões.

Existem diversas estratégias, mas vai-se resolver aqui a chamada busca (ou caminhamento – são sinônimos) em largura, cuja descrição é: *saindo de um vértice, visite todos os vértices a ele conectados, antes de passar a outro vértice. Repita isto até ter visitado todos os vértices do grafo.*

Algoritmo de Dijkstra

No algoritmo de Dijkstra (lê como Dícster) as arestas do grafo contém valores não negativos (preço, demora, vazão, capacidade, disponibilidades...). Este tipo de grafo é chamado grafo valorado. O algoritmo responde a pergunta: saindo de um vértice qual o caminho "mais barato" para chegar a algum outro vértice ? O algoritmo começa com uma lista dos vizinhos do vertice de saída com os custos associados a cada vizinho. Agora o algoritmo se desloca a cada vizinho e refaz essa busca. Se algum vizinho do vizinho tiver custo menor para ser alcançado esse custo é atualizado (para baixo). Quando o vértice final for alcançado o algoritmo acabou. Note-se que no grafo não podem haver arestas negativas nem ciclos.

na implementação abaixo, usou-se uma matriz de custos. É uma matriz quadrada de ordem igual ao número de vértices do grafo. Nela, o valor $C[i, j]$ corresponde ao custo para ir do nodo i ao nodo j.

```
def dijk(t,ini,fim):
  precede=numpy.zeros((len(t)))
  perm=numpy.zeros((len(t)))
  dist=numpy.zeros((len(t)))
  for i in range(len(dist)):
    dist[i]=9999
  perm[ini]=1
  dist[ini]=0
  corrente=ini
  while corrente != fim:
    menordist=9999
    dc=dist[corrente]
    for i in range(len(dist)):
      if perm[i]==0:
        novadist=dc+t[corrente][i]
        if novadist<dist[i]:
          dist[i]=novadist
          precede[i]=int(corrente)
        if dist[i]<=menordist:
          menordist=dist[i]
          k=i
    corrente=k
    perm[corrente]=1
  print(dist[fim])
  g=fim
  while ini !=g:
    print(int(g))
    g=precede[int(g)]
  print(ini)
import numpy
t=numpy.array([[9999,11,21,44,20,5,43,42,9],
               [8,9999,23,55,51,32,28,3,27],
               [2,18,9999,54,47,49,59,35,4],
               [17,26,10,9999,9,4,14,57,11],
               [7,13,29,16,9999,22,31,33,12],
               [1,2,1,38,10,9999,15,22,50],
               [53,6,16,18,34,8,9999,46,36],
               [13,39,45,3,7,5,48,9999,17],
               [19,40,30,25,52,20,24,41,9999]])
print(t)
dijk(t,0,3)
```

Algoritmo Guloso

Existem muitos problemas na ciência da computação cuja solução exata só é possível para instâncias muito pequenas do problema. No jargão são conhecidos como "problemas NP-completos". Um famoso é o problema do caixeiro viajante *(um caixeiro deve sair de uma cidade, visitar um conjunto de outras cidades e retornar à primeira, gastando o menor trajeto possível).* Para instâncias maiores é impossível resolver exatamente o problema. Nestes casos, precisa-se usar algum tipo de aproximação. Uma muito boa atende pelo nome de "estratégia gulosa"que pode ser jocosamente descrita como *em cada momento, abocanhe o maior pedaço.* Seja o seguinte problema: Você quer inaugurar um programa de rádio e quer atingir todos os estados brasileiros. Há um conjunto de rádios, cada uma com seus custos e sua área de abrangência. A pergunta é: qual o menor (mais barato) conjunto de rádios que atinge todos os estados brasileiros ? Veja uma instância possível do problema:

```
Radio     Estados que abrange    Custo
WQ1       PR, SC, SP, RS         1000/minuto
WQ2       SP, RJ                 800/minuto
WQ3       MG, SP, RJ, ES, BA     1100/minuto
...
```

Note que o problema está na sobreposição: praticamente todas as rádios transmitem para SP (o maior mercado) e é essa sobreposição que complica a solução. O algoritmo guloso neste caso fica:
1. Localize a rádio que abranja o maior número de estados ainda não cobertos. Não há problema se esta rádio atingir estados já cobertos. 2. Agregue esta rádio ao conjunto mínimo
3. repita o processo até ter todos os estados cobertos no conjunto mínimo.

Este algoritmo pode não achar a melhor resposta, mas sempre acha uma boa resposta e demora muito menos que o algoritmo exato. Compare os tempos

número deões estações	alg. exato $O(n!)$	alg. guloso $O(n^2)$
5	3.2 seg	2.5seg
10	102 seg	10 seg
32	13 anos	102 seg
100	4×10^{21} anos	16 minutos

☞ Para você fazer

Conduza uma pesquisa na Internet e faça um resumo pessoal dos seguintes algoritmos/sites:

1. Troca de Chaves Diffie-Hellman
2. Algoritmo HyperLogLog
3. Algoritmo Flajolet-Martin
4. Algoritmo Hash e SimHash (um hash que tem sensibilidade local)
5. o site better explained
6. o algoritmo de similaridade do cosseno
7. o algoritmo da maior subsequência comum
8. algoritmo da mochila (aplicação do alg. guloso)