MAC 328 Algoritmos em Grafos

Algoritmo de Bellman-Ford-Moore

Observação. A descrição do Algoritmo de Bellman-Ford-Moore que está abaixo foi extraída do livro:

R.E. Tarjan, *Data Structures and Network Algorithms*, CBMS-NSF Regional Conference Series in Applied Mathematics, Philadelphia, Pennysylvania, 1983.

Seja *D=(V,A)* um grafo e *s* um vértice de *D*. Antes de descrevermos o Algoritmo de Bellman-Ford-Moore é bom lembrarmos que nem sempre é possível encontrarmos um passeio mínimo de um vértice a todos os outros vértices de um grafo.

Teorema 1. Seja *s* um vértice de um grafo *D* tal que todos os vértices de *D* são acessíveis a partir de *s*. Então vale uma, e apenas uma, das seguintes afirmação:

- *D* possui uma árvore de caminhos mínimos com raiz *s*.
- *D* contém um ciclo negativo.

Se *T* é uma árvore (ou arborescência) geradora de *D* com raiz *s* e *v* é um vértice de *D*, nós definimos *dist[v]* como sendo a distância de *s* a *v* em *T*. O algoritmo que descreveremos é baseado no seguinte teorema.

Teorema 2. (Caracteriza��o de �rvores de caminhos m�nimos) *T* é uma árvore de caminhos mínimos se e somente se, para cada arco *(u,v)* de *D* vale que

$$dist[u] + len(u,v) >= dist[v].$$

O teorema acima sugeri a seguinte maneira para construirmos uma árvore de caminhos mínimos com raiz *s*. Para cada vértices *v* manteremos uma distância tentativa *dist[v]* de *s* até *v* e um predecssor tentativo *pred[v]* da árvore de caminhos mínimos. No início *dist[s] = 0* e *dist[v]* = `INFINITO` para os demais vértices de *D*. Além disso *pred[v]* = `NULL` para todos os vértices do grafo. Iteraremos o seguinte passo até que dist[u] + len(u,v) >= dist[v]. para todo arco *(u,v)* de *D*:

`LABELING STEP` (L.R. Ford). Selecione um arco *(u,v)* tal que *dist[u] + len(u,v) < dist[v]* . Faça

$dist[v] = dist[u] + len(u,v)$ e $pred[v] = u$.

O *labeling method* descrito acima não pára caso o grafo contenha um ciclo negativo. Ademais, mesmo que o grafo não contenha ciclos negativos e o método pare, ele pode demorar muito. Nosso primeiro refinamento será evitar que arcos sejam examinados desnecessariamente.

O *labeling and scanning method* mantém uma partição dos vértices em três estados: `NÃO-VISTO`, `ROTULADO` e `EXAMINADO`. Os vértices `ROTULADOS` e `EXAMINADOS` são exatamente aqueles que têm distância tentativa finita. Inicialmente *s* é `ROTULADO` e os demais vértices são `NÃO-VISTOS`. O método consiste em repetir o seguinte passo até que não hajam vértices `ROTULADOS`.

`SCANNING STEP`. Selecione um vértice `ROTULADO` *u* e examine-o, mudando o seu estado para `EXAMINADO`, através da aplicação do *labeling step* a cada arco *(u,v)* tal que $dist[u] + len(u,v) < dist[v]$ e convertendo o estado de *v* para `ROTULADO` (não importa se o estado de *v* era `NÃO-VISTO` ou `EXAMINADO`).

O *labeling e scanning method*, a exemplo do *labeling method*, é ineficiente e não pára caso o grafo contenha um ciclo negativo acessível a partir de *s*. Entretanto, podemos transformar este método em um algoritmo eficiente através de uma escolha apropriado da ordem em que os vértices `ROTULADOS` serão examinados.

E.F. Moore e, independentemente, R.E. Bellman propuseram que os vértices `ROTULADOS` fossem examinados de acordo com uma ordem de busca em largura (*breadth-first order*): entre os vértices `ROTULADOS` examine aquele que foi rotulado a mais tempo. Para implementar a busca em largura representamos o conjunto de vértices `ROTULADOS` através de uma fila. Nesta implementação surge a questão do que fazer com um vértice `ROTULADO` que é rotulado novamente. Um interpretação estrita de busca em largura requer que nós movimentemos um vértice rotulado para o final da fila. Nos usaremos uma interpretação mais livre, deixaremos o vértice rotulado na sua posição corrente na fila. O método abaixo implementa esta versão do *labeling and scanning method* usando busca em largura. Note que o método, como está descrito, ainda não pára se o grafo contém ciclos negativos.

```
1   BELLMAN-FORD-MOORE (Grafo D=(V,A), Vértice s)
2     para cada vértice v de D faça
3         estado[v] := NÃO-VISTO
4         dist[v] := INFINITO
5         pred[v] := NULL
6     estado[s] := ROTULADO
7     dist[s] := 0
8     Q := INICIALIZA-FILA(s)
9     enquanto Q não está vazia faça
10        u := PRIMEIRO-DA-FILA(Q)
11        para cada v em Adj[u] faça
```

```
12            se dist[u] + len(u,v) < dist[v] então
13              estado[v] := ROTULADO
14              dist[v] := dist[u]+len(u,v)
15              pred[v] := u
16              se v não está em Q então
17                INSIRA-NA-FILA(Q,v)
18        REMOVA-DA-FILA(Q)
          estado[u] := EXAMINADO
      devolva dist e pred
```

Ao longo do algoritmo, `Q` é uma *fila de vértices*; essa fila é o segredo do funcionamento do algoritmo. O comando INICIALIZA-FILA(`s`) cria uma fila com um só elemento igual a `s`. O comando PRIMEIRO-DA-FILA(`Q`) devolve o primeiro elemento da fila `Q` mas não retira esse elemento da fila. O comando INSIRA-NA-FILA(`Q,v`) insere `v` no fim da fila `Q`. O comando REMOVA-DA-FILA(`Q`) remove o primeiro elemento da fila `Q`.

Para analisarmos o dempenho deste método nós dividiremos a sua execução em passos. O passo zero consiste do exame do vértice *s*. Para $j > 0$, o passo *j* consiste do exame de todos os vértices na fila ao final do passo $j - 1$. Cada passo pode ser executado em tempo $O(m)$ pois durante cada passo cada arco é examinado no máximo uma vez. O teorema a seguir fornece um limite ao número total de passos.

Teorema 3. Se *D* não contém um ciclo negativo acessível a partir de *s* então examinar os vértices através de busca em largura gasta tempo $O(nm)$ e não mais do que $n-1$ passos são realizados. Caso contrário, o método não pára.

Prova. Podemos provar por indução em *k* que se existe um passeio mínimo de *s* até um vértice *v* usando *k* arcos, então no início do passo *k* a distância tentativa `dist[v]` será igual ao comprimento deste passeio mínimo.

Para transforma o método acima em um algoritmo nós devemos fazer com que ele pare mesmo na presença de ciclos negativos. Uma maneira fácil de fazer isto é contar passos. Nós incluímos duas variáveis ao método, um inteiro passo e uma váriavel último indicando o último vértice que foi examinado durante o presente passo do algoritmo. Inicialmente passo = 0 e último = s. Depois que o vértice último é examinado nós incrementamos de 1 a variável passo e último passa a ser o último vértice da fila Q. Se passo receber o valor n com a fila Q não vazia, o (agora) algoritmo deve parar e anunciar a presença de um ciclo negativo no grafo. Com este contador de passos algoritmo gasta tempo $O(nm)$ não importando se o grafo tem ou não um ciclo negativo. Usando i lema a seguir podemos localizar um ciclo negativo, se um tla ciclo existir.

Lema 4. Se a fila Q não está vazia no final do passo $n-1$ então $pred^k[v] = v$ para algum vértice v e inteiro k e o correspondente ciclo em D é negativo.

Prova. Suponha que o algoritmo é executado até que um vértice w seja examinado no passo n. Defina passo[v] de um vértice v como sendo o máximo j tal que v é examinado durante o passo j. Se passo[v] é definido e positivo

então pred[v] e passo[pred[v]] são definidos e passo[pred[v]] >= passo[v] - 1. Temos que passo[w]=n. Seguindo os apontadores pred a partir de W nos devemos, em algum momento, repetir um vértice, pois existem somente *n* vértices e o passo de cada vértice decresce de no máximo um em cada passo.

Detalhes de implementação

Você deve implementar o algoritmo de Bellman-Ford-Moore como uma função em seu programa, de forma a permitir que, no futuro, a mesma função possa ser usada inalterada por outros programas que você for escrever.

O algoritmo precisa de uma estrutura de dados auxiliar que permita as operações INICIALIZA-FILA, PRIMEIRO-DA-FILA, INSIRA-NA-FILA, e REMOVA-DA-FILA. A fila deverá ser representada como uma lista linear (vetor) de apontadores para Vertex.

Dê uma olhada no arquivo gb_dijk.w do Stanford GraphBase para ter uma idéia geral de uma implementação do Algoritmo Dijkstra. O arquivo miles_span.w contém uma implementação do heap com as operações que a gente precisa.

Grafo de cidades brasileiras

O professor José Augusto fez programas para criar grafos (.gb) de cidades brasileiras. Veja os dados, grafos e os programas em http://www.ime.usp.br/~coelho/grafos/sgb/cidades/. Você pode usar este grafo de cidades brasileiras para brincar com o seu programa.

Módulo GB_MILES

Este módulo do SGB contém a subrotina miles que você pode usar para testar e brincar com o seu programa bellman.

Copiado do arquivo gb_miles.w: The subroutine call

```
miles(n,north_weight,west_weight,pop_weight,max_distance,max_degree,seed)
```

constructs a graph based on the information in miles.dat. Each vertex of the graph corresponds to one of the 128 cities whose name is alphabetically greater than or equal to `Ravenna, Ohio' in the 1949 edition of Rand McNally & Company's *Standard Highway Mileage Guide*. Edges between vertices are assigned lengths representing distances between cities, in miles. In most cases these mileages come from the Rand McNally Guide, but several dozen entries needed to be changed drastically because they were obviously too large or too small; in such cases an educated guess was made. Furthermore, about 5% of the entries were adjusted slightly in order to ensure that all distances satisfy the ``triangle inequality'': The graph generated by miles has the property that the distance from *u* to *v* plus the distance from *v* to *w* always exceeds or equals the distance from *u* to *w*.

The constructed graph will be ``complete''---that is, it will have edges between every pair of vertices---unless special values are given to the

parameters max_distance or max_degree. If max_distance!=0, edges with more than max_distance miles will not appear; if max_degree!=0, each vertex will be limited to at most max_degree of its shortest edges.

Um exemplo de uso da função miles pode ser encontrado em gb_span.w, como mostrado abaixo.

```
#include "gb_miles.h" /* the |miles| routine */
[...]
main(argc,argv)
  int argc; /* the number of command-line arguments */
  char *argv[]; /* an array of strings containing those arguments */
{@+unsigned long n=100; /* the desired number of vertices */
  unsigned long n_weight=0; /* the |north_weight| parameter */
  unsigned long w_weight=0; /* the |west_weight| parameter */
  unsigned long p_weight=0; /* the |pop_weight| parameter */
  unsigned long d=10; /* the |max_degree| parameter */
  long s=0; /* the random number seed */
  unsigned long r=1; /* the number of repetitions */
  char *file_name=NULL; /* external graph to be restored */
  @;
  while (r--) {
    if (file_name) g=restore_graph(file_name);
    else g=miles(n,n_weight,w_weight,p_weight,0L,d,s);
    if (g==NULL || g->n<=1) {
      fprintf(stderr,"Sorry, can't create the graph! (error code %ld)\n",
               panic_code); /* panic_code is set nonzero if graph
                             * generator panics.
                             * panic_code indica o problema que ocorreu,
                             * veja gb_graph.w
                             */
      return -1; /* error code 0 means the graph is too small */
    }
```

Leia mais sobre este módulo em gb_miles.w.

Módulo GB_RAND

Copiado do arquivo gb_rand.w: This GraphBase module provides two external subroutines called random_graph and random_bigraph, which generate graphs in which the arcs or edges have been selected ``at random.'' A third subroutine, random_lengths, randomizes the lengths of the arcs of a given graph. The performance of algorithms on such graphs can fruitfully be compared to their performance on the nonrandom graphs generated by other GraphBase routines.

[...]

The procedure

```
random_graph(n,m,multi,self,directed,dist_from,dist_to, min_len,max_len,seed)
```

is designed to produce a pseudo-random graph with n vertices and m arcs or

```
edges, using pseudo-random numbers that depend on seed in a system-
independent fashion. The remaining parameters specify a variety of options:
multi!=0 permits duplicate arcs;
self!=0 permits self-loops (arcs from a vertex to itself);
directed!=0 makes the graph directed; otherwise each arc becomes an
undirected edge;
dist_from and dist_to specify probability distributions on the arcs;
min_len and max_len bound the arc lengths, which will be uniformly
distributed between these limits.

If dist_from or dist_to are NULL, the probability distribution is uniform
over vertices; otherwise the dist parameter points to an array of n
nonnegative integers that sum to 2^30, specifying the respective probabilities
(times 2^30) that each given vertex will appear as the source or destination of
the random arcs.

[...]
```

Conselho

Brinque com o seu programa, brinque *muito*. Aplique-o ao grafo de cidades brasileiras e a vários grafos do Stanford GraphBase (use as funções miles e random_graph. Durante a fase de testes, use grafos pequenos. Faça um desenho do grafo num pedaço de papel para conferir a resposta do seu programa.

Uso de WEB

Queria destacar que o uso do WEB, segundo o seu criador D. Knuth, tem as seguintes finalidades de

- escrever programas de qualidade superior;
- produzir *state-of-the-art documentation*;
- reduzir tempo de *debugging*;
- fazer facilmente a manutenção de programas.

Como conseqüência, Knuth comenta que fica mais divertido escrever programas.

Mudando de assunto, um comentário sobre comentários: ``Antes de cada função [e bloco] diga **o que** a função [e bloco] faz. Procure responder as perguntas que um usuário faria: que dados essa função recebe? que suposições faz sobre esses dados? que resultados devolve? Não perca tempo tentando dizer **como** a função faz o que ela faz.''(Paulo Feofiloff) Se o tamanho de seus blocos é razoavelmente pequeno, como deve ser, o conselho é extremamente útil.

Observação. Cópiei o comentário acima da página de grafos do prof. José Augusto

---

[MAC 328's Home Page. | Exercícios-programas]

Last modified: Wed Oct 13 14:04:13 EDT 1999