Exemplo de uso do Stanford GraphBase com CWEB-LaTeX

Busca em Profundidade

coelho

19 de março de 2003

Sumário

1. Introdução

Existem pelo menos duas maneiras populares de se fazer busca sistemática em um grafo: busca em largura (*breadth-first search* e busca em profundidade (depth-first search). Neste programa implementamos a versão recursiva de busca em profundidade.

2. O que o programa faz

Este programa recebe um grafo, através de um arquivo no formato do `SGB` ou de uma opção para gerá-lo, e constrói uma floresta de busca em profundidade do grafo.

3. Como usar o programa

Para usar o programa o usuário deve, na linha de comando, escrever

`meu_prompt> dfs` ⟨`opções`⟩

onde ⟨`opções`⟩ consiste de zero ou mais das seguintes especificações em qualquer ordem

`-r` Usa a subrotina *random_graph* do módulo GB_RAND do `SGB` para gerar um grafo pseudo-aleatório com n vértice m arcos ou arestas, usando um gerador de números pseudo-aleatórios que depende de um semente (veja opção `-s`).

`-c` Usa a subrotina *circuit* do módulo GB_BASIC do `SGB` para gerar um circuito simétrico com n vértices. Todo circuito é hamiltoniano.

`-e` Usa a subrotina *empty* do módulo GB_BASIC do `SGB` para gerar um grafo com n vértices e zero arcos. O grafo sem arcos não é hamiltoniano.

`-p` Usa a subrotina *petersen* do módulo GB_BASIC do `SGB` para gerar o grafo de Petersen. O grafo de Petersen não é hamiltoniano.

`-t` Usa a subrotina *transitive* do módulo GB_BASIC do `SGB` para gerar um torneio com n vértices. Um torneio é um grafo que tem um arco de u a v se $u < v$. Torneios não são hamiltonianos.

`-k` Usa a subrotina *complete* do módulo GB_BASIC do `SGB` para gerar um grafo simétrico completo com n vértices. Num grafo completo simétrico temos uma arco de u a v para cada para u e v de vértices distintos. Todo grafo completo simétrico é hamiltoniano.

`-g`⟨`nome`⟩ Nenhum grafo será gerado. O grafo sobre o qual o programa trabalhará será lido do arquivo ⟨nome⟩.

`-s`⟨`número`⟩ Usa o número s como semente do gerardor de números pseudo-aleatórios, onde s é o ⟨número⟩ dado.

`-n`⟨`número`⟩ O grafo gerado terá n vértices, onde n é o ⟨número⟩ dado.

`-m`⟨`número`⟩ O grafo gerado terá m arcos ou arestas, onde m é o ⟨número⟩ dado.

`-h`⟨`número`⟩ Usa a heurística do vértice de grau de saída mínimo.

`-o`⟨`nome`⟩ O grafo de trabalho é salvo no arquivo ⟨nome⟩.

É evidente que só faz sentido usar apenas uma entre as opções `-r, -c, -e, -p, -t, -k, -g`. Também não faz sentido usar a opção `-g` junto com as opções `-n, -s` e `-m`. Só faz sentido usar a opção `-m` junto com `-r`.

Como de hábito, caso o usuário escreva apenas

`meu_prompt> dfs`

o programa deve descrever o seu funcionamento.

⟨ Função *mostre_uso* 3 ⟩ ≡

```
void mostre_uso()
{
  fprintf(stdout,
  "Uso:␣%s␣<opcoes>\n"
  "onde␣<opcoes>␣consiste␣de␣zero␣ou␣mais␣das␣seguintes␣especificações:\n"
  "␣␣␣␣␣-r␣␣grafo␣pseudo-aleatorio␣com␣n␣vertices␣e␣m␣arestas\n"
  "␣␣␣␣␣-c␣␣circuito␣com␣n␣vertices\n"
  "␣␣␣␣␣-e␣␣grafo␣com␣n␣vertices␣e␣zero␣arcos\n"
  "␣␣␣␣␣-p␣␣grafo␣de␣Petersen\n"
  "␣␣␣␣␣-t␣␣torneio␣com␣n␣vertices\n"
  "␣␣␣␣␣-k␣␣grafo␣simetrico␣completo␣com␣n␣vertices\n"
  "␣␣␣␣␣-g<nome>␣trabalha␣com␣o␣grafo␣do␣arquivo␣<nome>\n"
```

```
"␣␣␣␣␣-s<num>␣␣para␣o␣gerar␣numeros␣pseudo-aleatorios\n"
"␣␣␣␣␣-n<num>␣numero␣de␣vertices\n"
"␣␣␣␣␣-m<num>␣numero␣de␣arcos/arestas\n"
"␣␣␣␣␣-o<nome>␣grafo␣de␣trabalho␣sera␣salvo␣no␣arquivo␣<nome>\n"
, nome_programa);
exit(0);
}
```

Este código é usado no bloco 6.

4. Estrutura geral do programa

Abaixo encontra-se o 'layout' do programa, como é visto pelo compilador `C`.

```
#include <stdio.h>
#include "gb_graph.h"     /* estruturas de dados padrão do SGB */
#include "gb_save.h"      /* rotinas para ler e salvar um grafo */
#include "gb_basic.h"     /* rotinas para gerar alguns grafos básicos */
#include "gb_rand.h"      /* produz um grafo pseudo-aleatório */
  ⟨ Preprocessor definitions ⟩
  ⟨ Variáveis globais 5 ⟩
  ⟨ Declaração de funções 6 ⟩
  int main(argc, argv)
      int argc;     /* o número de argumentos na linha de comando */
      char *argv[];     /* o vetor de strings contendo os argumentos */
  {
    ⟨ Variáveis locais do main 7 ⟩;
    ⟨ Examine as opções na linha de comando 8 ⟩;
    ⟨ Crie o grafo g dado 10 ⟩;
    ⟨ Construa uma floresta dfs de g 15 ⟩;
    if (salva_arquivo) ⟨ Salve o grafo g no arquivo salva_arquivo 23 ⟩;
    gb_recycle(g);     /* devolve a memória usada por g */
    return 0;     /* termino normal */
  }
```

5. Gosto que *nome_programa* e algumas outras variáveis sejam globais.

```
⟨ Variáveis globais 5 ⟩ ≡
  typedef enum {
    FALSE, TRUE
  } Boolean;
  char *nome_programa = Λ;     /* nome do programa */
```

Veja também blocos 11 e 12.

Este código é usado no bloco 4.

6. A seguir está a lista de funções que fazem parte do programa.

```
⟨ Declaração de funções 6 ⟩ ≡
  ⟨ Função mostre_uso 3 ⟩
  ⟨ Função dfs_r 16 ⟩
```

Este código é usado no bloco 4.

7. Tratamento das opções

Entre as váriaveis local do *main* teremos o grafo g sobre o qual trabalharemos. Outras variáveis são relativas a opções do programa. Estas váriáveis são inicializadas com seu valores default. Assim

```
meu_prompt> dfs -n30
```

tem o mesmo efeito que

```
meu_prompt> dfs -r -n30 -m100
```

⟨ Variáveis locais do *main* 7 ⟩ ≡

```
unsigned long n = 20;    /* número de vértices */
unsigned long m = 100;    /* número de arcos */
long directed = 1;    /* o grafo gerado não precisa ser simétrico */
long seed = 0;    /* semente para o gerador pseudo-aleatório */
Boolean aleatório = FALSE;    /* gerar um grafo pseudo-aleatório */
Boolean circuito = FALSE;    /* gerar um circuito */
Boolean vazio = FALSE;    /* gerar um grafo sem arcos */
Boolean petersen = FALSE;    /* gerar o grafo de Petersen */
Boolean torneio = FALSE;    /* gerar um torneio */
Boolean completo = FALSE;    /* gerar um grafo completo */
Boolean heurística = TRUE;    /* heurística do grao de saída mínimo */
Boolean escolheu = FALSE;    /* indica se foi escolhido como criar o grafo */
char *nome_arquivo = Λ;    /* nome do arquivo do grafo de trabalho */
char *salva_arquivo = Λ;    /* para salvar o grafo de trabalho */
```

Veja também bloco 9.

Este código é usado no bloco 4.

8. As opções podem ser fornecidas em qualquer ordem. Se o grafo é dado por mais de uma maneira (opções `-r`, `-c`, `-e`, `-p`, `-t`, `-k` ou `-g`), o programa pára com uma mensagem de erro. A variável *escolheu* é usada para detectar este tipo de inconsistência. Está é a única inconsistência verificada na linha de comando. Assim, por exemplo, o programa não reclama se for chamado com as opções `-c -n10 -m1000`; note que todo circuito com 10 vértices tem 20 arcos.

⟨ Examine as opções na linha de comando 8 ⟩ ≡

```
nome_programa = argv[0];
if (argc ≡ 1) mostre_uso();
while (−− argc) {
  if (strcmp(argv[argc], "-r") ≡ 0 ∧ ¬escolheu) escolheu = aleatório = TRUE;
  else if (strcmp(argv[argc], "-c") ≡ 0 ∧ ¬escolheu) escolheu = circuito = TRUE;
  else if (strcmp(argv[argc], "-e") ≡ 0 ∧ ¬escolheu) escolheu = vazio = TRUE;
  else if (strcmp(argv[argc], "-p") ≡ 0 ∧ ¬escolheu) escolheu = petersen = TRUE;
  else if (strcmp(argv[argc], "-t") ≡ 0 ∧ ¬escolheu) escolheu = torneio = TRUE;
  else if (strcmp(argv[argc], "-k") ≡ 0 ∧ ¬escolheu) escolheu = completo = TRUE;
  else if (strncmp(argv[argc], "-g", 2) ≡ 0 ∧ ¬escolheu) {
    escolheu = TRUE;
    nome_arquivo = argv[argc] + 2;
  }
  else if (strcmp(argv[argc], "-h") ≡ 0) heurística = TRUE;
  else if (sscanf(argv[argc], "-n%lu", &n) ≡ 1) ;
  else if (sscanf(argv[argc], "-m%lu", &m) ≡ 1) ;
  else if (sscanf(argv[argc], "-s%ld", &seed) ≡ 1) ;
  else if (strncmp(argv[argc], "-o", 2) ≡ 0) salva_arquivo = argv[argc] + 2;
  else {
```

```
    fprintf(stderr, "Uso:␣%s␣[-r|-c|-e|-p|-t|-k|-g<nome>]]␣[-h]␣[-s<semente>]␣[-\
        n<num>]␣[-m<num>]␣[-o<nome>]\n", nome_programa);
    return −1;
  }
}
if (¬escolheu) aleatório = TRUE;     /* opção default */
```

Este código é usado no bloco 4.

9. Criação do grafo

O trabalho desta parte do programa é construir ou reconstruir o grafo g sobre o qual trabalharemos. O grafo será construído se algumas das opções de geração automática do grafo foi usada. Caso a opção `-g⟨nome⟩` tenha sido usada o grafo será recontruído a partir do conteúdo de *nome_arquivo*. A criação do grafo depende da

```
⟨ Variáveis locais do main 7 ⟩ +≡
  Graph *g = Λ;    /* grafo de trabalho */
```

10. Se uma rotina de criação do grafo encontrar um problema, ela devolve Λ (ou seja, `NULL`), depois de colocar um número de código que identifica o erro na variável externa *panic_code*. Se não ocorrer problemas a rotina devolve o endereço do grafo criado (**Graph** ∗).

```
⟨ Crie o grafo g dado 10 ⟩ ≡
  if (aleatório) g = random_graph(n, m, 0, 0, directed, Λ, Λ, 1, 1, seed);
  else if (circuito) g = circuit(n);
  else if (vazio) g = empty(n);
  else if (petersen) g = petersen();
  else if (torneio) g = transitive(n);
  else if (completo) g = complete(n);
  else if (nome_arquivo) g = restore_graph(nome_arquivo);
  if (g ≡ Λ)
  {
    fprintf(stderr, "%s: grafo nao pode ser criado (codigo do erro: %ld)\n", nome_programa,
        panic_code);
    return −1;    /* Opsss ... */
  }
  fprintf(stdout, "grafo: %s.\nnumero de vertices: %ld.\nnumero de arcos: %ld.\n", g→id, g→n, g→m);
```

Veja também bloco 14.

Este código é usado no bloco 4.

11. Busca em profundidade

Finalmente chegamos ao coração do programa. Durante a execução do programa cada vértice do grafo está no estado `NÃOVISTO`, `VISITADO` ou `EXAMINADO`. Um vértice u está no estado `NÃOVISTO` se programa ainda não encontro um caminho de r até u. Os vértices `VISITADO` fazem parte de um caminho na iteração corrente de um vértice raiz r a um vértice u. Um vértice u está no estado `EXAMINADO` se todos os arcos com ponto inicial em u já foram examinados.

A estruturas de dados que representa o estado corrente do algoritmo é implementada usando três 'utility fields' em cada vértice: *estado*, *pred* (predecessor). Além disso ainda há um campo que guarda o número *pré_ordem* do vértices. Consideraremos cada um deles separadamente.

#define *estado* $z.I$ /* o *estado* de um vértice é armazenado no utility field z */

⟨ Variáveis globais 5 ⟩ +≡
enum {
`NÃOVISTO` = 0, `VISITADO` = 1, `EXAMINADO` = 2
};

12. Os vértices no estado `VISITADO` formam um caminho de um vértice raiz r a um vértice u. Todo vértice `VISITADO` tem um predecessor no caminho, indicado pelo campo *pred*. Este predecessor é outro vértice ativo; a única excessão é a raiz r que tem ele mesmo como predecessor. Assim, um arco de v a w no caminho é representado na estrutura por $w\to pred \equiv v$. O campo *pred* permite que o programa faça backtracking no caminho.

Os campos *pred* mantém a chamada *arvoré de busca em profundidade*. Assim, no fundo, o programa mantém uma representação 'reversa' do caminho corrente de r a cada vértice u na árvore de busca emprofundidade, ou seja, o caminho de r a um vértice u `VISITADO` ou `EXMINADO`.

$$\langle s = u\to pred \to \ldots \to pred\ , \ldots, u\to pred\to pred\to pred, u\to pred\to pred, u\to pred, u\rangle.$$

#define *pred* $y.V$ /* o predecessor (*pred*) de um vértice e armazenado no utility field x */
#define *pré_ordem* $x.I$ /* número pré-ordem do vértice */
#define *pós_ordem* $w.I$ /* número pós-ordem do vértice */

⟨ Variáveis globais 5 ⟩ +≡
long *préordem* = 0;
long *pósordem* = 0;

13. Busca em profundidade explora um grafo sistematicamente, visitando todos os vértices e verificando para onde estes vértices podem nos conduzir. O vértice no estado `VISITADO` que é ponta final do caminho corrente é u.

Há três possibilidade, o arco a com ponta inicial em u tem ponta final $a\to tip$ em um vértice v que é `NÃOVISTO`, `VISITADO` ou `EXAMINADO`. Se o vértice v é `VISITADO`, então v já faz parte do caminho corrente e tentaremos estender o caminho através do arco sucessor de a na lista de adjacência de u. O mesmo é feito se v é `EXAMINADO`. Caso v seja `NÃOVISTO`, marcamos v como `VISITADO` e v passa a ser a nova ponta final do caminho corrente e passamos a examinar os arcos com ponta inicial nele. Se chegarmos a um ponto em que todos os arcos com ponta inicial em u nos levam a vértices `VISITADOS` ou `EXAMINADOS`, chegou a hora de dar um passo atrás e fazer com que $u\to pred$ volte a ser a ponta final do caminho.

14. O campo *util_types* de um grafo contém um string de comprimento 14, seguido do caractere nulo '$T0$'. Os primeiros 6 caracteres do *util_type* indicam o uso dos campos utilitários u, v, w, x, y e z dos registros **Vertex**. Os próximos dois caracteres dão o formato dos campos utilitários a e b em **Arc**. Finalmente, os seis último caracteres mostram o formato dos campos utilitários uu, vv, ww, xx, yy e zz em **Graph**. Cada caractere deve ser um `I` (denotando um inteiro longo), `S` (representando o endereço de um string), `V` (indicando o endereço de um **Vertex**), `A` (denotando o endereço de um **Arc**), `G` (endereço de um **Graph**), ou `Z` (significando que o campo não está em uso).

⟨ Crie o grafo g dado 10 ⟩ +≡
strcpy($g\to util\_types$, "ZZIIVIZZZZZZZZ");

15. Construção da floresta dfs

⟨ Construa uma floresta dfs de g 15 ⟩ ≡

```
{
  register Vertex *v;
  for (v = g->vertices; v < g->vertices + g->n; v++)
  {
    v->estado = NÃOVISTO;
    v->pred = v;
  }
  for (v = g->vertices; v < g->vertices + g->n; v++)
    if (v->estado ≡ NÃOVISTO)
      dfs_r(g, v);
}
```

Este código é usado no bloco 4.

16. Função *dfs_r*.

⟨ Função *dfs_r* 16 ⟩ ≡

```
void dfs_r(Graph *g, Vertex *r)
{
  register Arc *a;
  ⟨ Visita em pré-ordem de r 22 ⟩;
  r->pré_ordem = ++préordem;
  r->estado = VISITADO;
  for (a = r->arcs; a; a = a->next)
  {     /* examina o arco a */
    Vertex *v = a->tip;
    if (v->estado ≡ NÃOVISTO)
    {
      ⟨ Trata o tree arc rv 17 ⟩;
      dfs_r(g, v);
      v->pred = r;      /* pred guarda a árvore dfs */
    }
    else if (v->estado ≡ VISITADO)
    {     /* achou um ciclo */
      ⟨ Trata o back arc rv 18 ⟩;
    }
    else if (v->estado ≡ EXAMINADO ∧ r->pré_ordem > v->pré_ordem) {
      ⟨ Trata o forward arc rv 19 ⟩;
    }
    else      /* v->estado ≡ EXAMINADO ∧ r->pré_ordem < v->pré_ordem */
    {
      ⟨ Trata o cross arc rv 20 ⟩;
    }
  }
  r->estado = EXAMINADO;
  ⟨ Visita em pós-ordem de r 21 ⟩;
  r->pós_ordem = ++pósordem;
}
```

Este código é usado no bloco 6.

17. Isto é só para não dar erro de sintaxe. Pode ser necessário dependendo da aplicação.

$\langle$ Trata o tree arc *rv* 17 $\rangle \equiv$

{ }

Este código é usado no bloco 16.

18. Isto é só para não dar erro de sintaxe. Pode ser necessário dependendo da aplicação.

$\langle$ Trata o back arc *rv* 18 $\rangle \equiv$

{ }

Este código é usado no bloco 16.

19. Isto é só para não dar erro de sintaxe. Pode ser necessário dependendo da aplicação.

$\langle$ Trata o forward arc *rv* 19 $\rangle \equiv$

{ }

Este código é usado no bloco 16.

20. Isto é só para não dar erro de sintaxe. Pode ser necessário dependendo da aplicação.

$\langle$ Trata o cross arc *rv* 20 $\rangle \equiv$

{ }

Este código é usado no bloco 16.

21. Isto é só para não dar erro de sintaxe. Pode ser necessário dependendo da aplicação.

$\langle$ Visita em pós-ordem de *r* 21 $\rangle \equiv$

```
{
  fprintf(stdout, "%s\n", r->name);
}
```

Este código é usado no bloco 16.

22. Isto é só para não dar erro de sintaxe. Pode ser necessário dependendo da aplicação.

$\langle$ Visita em pré-ordem de *r* 22 $\rangle \equiv$

{ }

Este código é usado no bloco 16.

23. Salva grafo

Se o usuário utilizou a opção `-o`⟨nome⟩ o grafo g é salvo no arquivo ⟨nome⟩. É usual que o arquivo que contém um grafo no formato do `SGB` tenha a extensão `.gb`. Para salvar um grafo usamos a subrotina *gb_save* do módulo GB_SAVE do `SGB`.

⟨ Salve o grafo g no arquivo *salva_arquivo* 23 ⟩ ≡

```
{
  long anomalia = save_graph(g, salva_arquivo);
  if (anomalia)
  {
    fprintf(stderr, "%s:␣grafo␣nao␣pode␣ser␣salvo␣no␣arquivo""␣%s␣(codigo␣do␣erro:␣%ld)\n",
        nome_programa, salva_arquivo, anomalia);
    gb_recycle(g);
    return −2;    /* Opsss ... */
  }
}
```

Este código é usado no bloco 4.

Índice Remissivo

Aqui vai uma lista de identificadores usados e onde eles aparecem. Entradas grifadas indicam os lugares de definições. Mensagens de erros também são mostradas.

Lista de Refinamentos

⟨ Construa uma floresta dfs de g 15 ⟩ Usado no bloco 4.
⟨ Crie o grafo g dado 10, 14 ⟩ Usado no bloco 4.
⟨ Declaração de funções 6 ⟩ Usado no bloco 4.
⟨ Examine as opções na linha de comando 8 ⟩ Usado no bloco 4.
⟨ Função *dfs_r* 16 ⟩ Usado no bloco 6.
⟨ Função *mostre_uso* 3 ⟩ Usado no bloco 6.
⟨ Salve o grafo g no arquivo *salva_arquivo* 23 ⟩ Usado no bloco 4.
⟨ Trata o back arc *rv* 18 ⟩ Usado no bloco 16.
⟨ Trata o cross arc *rv* 20 ⟩ Usado no bloco 16.
⟨ Trata o forward arc *rv* 19 ⟩ Usado no bloco 16.
⟨ Trata o tree arc *rv* 17 ⟩ Usado no bloco 16.
⟨ Variáveis globais 5, 11, 12 ⟩ Usado no bloco 4.
⟨ Variáveis locais do *main* 7, 9 ⟩ Usado no bloco 4.
⟨ Visita em pós-ordem de r 21 ⟩ Usado no bloco 16.
⟨ Visita em pré-ordem de r 22 ⟩ Usado no bloco 16.