Exemplo de uso do Stanford GraphBase com CWEB-LaTeX

Grafos Hamiltonianos

coelho

27 de outubro de 2004

Sumário

1. Introdução

Esta é uma demonstração do `SGB` com `CWEB`-LaTeX. Antes de descrever o problema e o programa para resolvê-lo é conveniente introduzir alguma terminologia.

Um *passeio* num grafo g é qualquer seqüência da forma

$$\langle v_0, a_1, v_1, \ldots, a_k, v_k \rangle \tag{1}$$

onde $v_0, \ldots, v_k$ são vértices, $a_1, \ldots, a_k$ são arcos e, para cada i, a_i é o arco $v_{i-1}v_i$. O vértice v_0 é o *início* do passeio e o v_k é seu *término*.

Um *caminho* é um passeio sem vértices repetidos. Já um *circuito* é um caminho onde o início e término coincidem. Um caminho ou circuito que contém todos os vértices do grafo é dito *hamiltoniano*. O *problema do circuito hamiltoniano* (*Hamiltonian circuit problem*), denotado pela sigla PCH é o seguinte:

Problema PCH(g): Dado um grafo g, encontrar um circuito hamiltoniano.

Um grafo que possui um circuito hamiltoniano é dito *hamiltoniano*. O PCH é um problema computacionalmente difícil. Tecnicamente falando o PCH é NP-difícil. Isto significa que todo algoritmo conhecido só consegue resolver o problema para grafos 'grandes' após consumir uma quantidade de tempo indecente. `MAC0338` Análise de Algoritmos é uma disciplina que esclarece o significado desse termo de uma maneira mais precisa.

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 decide se o grafo dado é ou não hamiltomiano.

Se o grafo é hamiltoniano o programa devolve, para a saída padrão, uma seqüência dos vértices do grafo que formam um circuito hamiltoniano. Já, se o grafo não é hamiltoniano, o programa responde apenas que lamenta ... Seria bacana se tivéssemos um programa que devolvesse algum objeto que fosse uma prova cabal de que o grafo não é hamiltoniano. Mais ainda, seria ótimo que o programa pudesse verificar rapidamente essas provas ou certificados. Uma característica dos problemas NP-difíceis é que estes admitem um certificado 'sucinto' (ou seja, verificável rapidamente) para a resposta SIM (no caso, o grafo é hamiltoniano), mas não se conhece um certificado sucinto para a respotas NÃO (no caso, o grafo não é hamiltoniano).

3. Como usar o programa

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

meu_prompt> hamiltoniano ⟨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> hamiltoniano

o programa deve descrever o seu funcionamento.

⟨ Função *mostre_uso* 3 ⟩ ≡

```
void mostre_uso()
{
  fprintf(stdout,
  "Uso:␣%s␣[-r|-c|-e|-p|-t|-k|-g<nome>]␣[-h]␣[-s<num>]␣[-n<num>]␣""[-m<num>]␣[-o<nome>]\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 7.

4. Estratégia de solução

Para resolver certos problemas é necessário enumerar todas as configurações (ou objetos) de um certo tipo. O número de configurações é tipicamente muito grande, e portanto os algoritmos enumerativos usualmente consomem uma quantidade absurda de tempo. Este programa é uma implementação de uma algoritmo enumerativo do tipo *backtracking*. Este tipo de algoritmo também é conhecido como *tipo força-bruta*, pois examina sistematicamente todos os candidados a solução. Você já estudou algoritmos enumerativos em `MAC0122`. Veja, por exemplo, o *problema das* n *rainhas*

`http://www.ime.usp.br/~coelho/mac122/conteudo/backtracking/nrainhas.c`

e do *problema do escalonamento*

`http://www.ime.usp.br/~coelho/mac122/eps/ep4/` .

Algoritmos enumerativos são muitas vezes escritos de maneira recursiva. Este programa, só por nos diversão, apresenta uma implementação iterativa.

5. Estrutura geral do programa

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

```
#include <stdio.h>
#include <string.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 6 ⟩
  ⟨ Declaração de funções 7 ⟩
  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 8 ⟩;
    ⟨ Examine as opções na linha de comando 9 ⟩;
    ⟨ Crie o grafo g dado 11 ⟩;
    ⟨ Verifique se g é hamiltoniano 18 ⟩;
    if (salva_arquivo) ⟨ Salve o grafo g no arquivo salva_arquivo 24 ⟩;
    gb_recycle(g);     /* devolve a memória usada por g */
    return 0;     /* termino normal */
  }
```

6. Gosto que *nome_programa* e algumas outras variáveis sejam globais. Além disso, se o programa encontra algum problema, então um código de erro é devolvido através de *exit*.

```
⟨ Variáveis globais 6 ⟩ ≡
  typedef enum {
    FALSE, TRUE
  } Boolean;
  char *nome_programa = Λ;     /* nome do programa */
  enum {     /* código de erros */
    opção_invalida = −1,
    salva_grafo_erro = −2,
    putz_programa_errado = −3
  };
```

Veja também bloco 13.

Este código é usado no bloco 5.

7. A seguir está a lista de funções que fazem parte do programa. Hummm, neste programa só temos uma função, excluindo-se o *main*.

```
⟨ Declaração de funções 7 ⟩ ≡
  ⟨ Função mostre_uso 3 ⟩
```

Este código é usado no bloco 5.

8. 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> hamiltoniano -n30
```

tem o mesmo efeito que

```
meu_prompt> hamiltoniano -r -n30 -m100
```

⟨ Variáveis locais do *main* 8 ⟩ ≡

```
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 = FALSE;   /* 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 blocos 10, 14 e 22.

Este código é usado no bloco 5.

9. 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 9 ⟩ ≡

```
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);
    exit(opção_invalida);
  }
}
if (¬escolheu) aleatório = TRUE;    /* opção default */
```

Este código é usado no bloco 5.

10. 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* 8 ⟩ +≡
```
Graph *g = Λ;      /* grafo de trabalho */
```

11. 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 11 ⟩ ≡
```
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 17.
Este código é usado no bloco 5.

12. Algoritmo enumerativo

Finalmente chegamos no coração do programa. Durante a execução do programa cada vértice do grafo está no estado 'ativo' ou 'inativo'. Os vértices ativos fazem parte de um caminho na iteração corrente de um vértice s a um vértice u e os vértices inativos são os que não estão no caminho. Grosseiramente falando, a missão da iteração é estender esse caminho corrente a um caminho hamiltoniano de s a um certo vértice t ou se arrepender-se (*backtrack*) e voltar nas suas 'próprias pegadas'.

A estruturas de dados que representa o estado corrente do algoritmo é implementada usando três 'utility fields' em cada vértice: *estado*, *pred* (predecessor) e *current_arc*. Consideraremos cada um deles separadamente.

13. O *estado* de um vértice é um inteiro. Ele é zero (ou, `ATIVO`) quando o vértice está inativo. Assim que o vértice passa a ser a ponta final u do caminho corrente ele se torna ativo e o valor de seu *estado* passa a ser um (ou `ATIVO`). Ao final do programa, caso um caminho hamiltoniano de s a t tenha sido encontrado, o programa certifica-se do seu achado e durante este processo transforma o estado dos vértices de `ATIVO` para `MARCADO`.

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

⟨ Variáveis globais 6 ⟩ +≡

```
enum {
  INATIVO = 0, ATIVO = 1, MARCADO = 2
};
```

14. Os vértices ativos formam um caminho de um vértice inicial s a um vértice u. Todo vértice ativo tem um predecessor no caminho, indicado pelo campo *pred*. Este predecessor é outro vértice ativo; a única excessão é a ponta inicial s do caminho que tem como predecessor o vértice especial t, que não é ativo. Assim, um arco de v a w no caminho é representado na estrutura por $w\text{→}pred \equiv v$. O campo *pred* permite que o programa faça backtracking no caminho.

Assim, no fundo, o programa mantém uma representação 'reversa' do caminho corrente de s a u, definida pelos campos *pred* de cada vértice ativo, ou seja, o caminho corrente é

$$\langle s = u\text{→}pred\text{→}\ldots\text{→}\ pred\ ,\ldots,\ u\text{→}pred\text{→}pred\text{→}pred,\ u\text{→}pred\text{→}pred,\ u\text{→}pred,\ u\rangle.$$

#**define** *pred* $y.V$ /* o predecessor (*pred*) de um vértice e armazenado no utility field y */

⟨ Variáveis locais do *main* 8 ⟩ +≡

```
Vertex *s;   /* vértice que é ponta inicial do caminho corrente */
Vertex *u;   /* vértice que é ponta final do caminho corrente */
Vertex *t;   /* vértice destino do caminho corrente */
```

15. Todos os arcos do grafo são explorados sistematicamente, a partir do vértice origem s, como numa busca em profundidade. Suponha que no caminho corrente existe um arco de v a w. Logo, $w\text{→}pred = v$. O arco de v a w também é representado diretamente na estrutura pelo campo *current_arc* do vértice v. Desta forma, tem-se que $v\text{→}current\_arc\text{→}tip = w$. Este tipo de apontador é bem conhecido e útil (veja o livro *Network Flows: Theory, Algorithms, and Applications*, R.K. Ahuja, Th.L. Magnanti, J.B. Orlin, Prentice Hall, 1993). O caminho corrente é representado por

$$\langle s,\ s\text{→}current\_arc\text{→}tip,\ s\text{→}current\_arc\text{→}tip\text{→}current\_arc\text{→}tip\ ,\ldots,\ s\text{→}current\_arc\text{→}tip\text{→}\ldots\text{→}\ current\_arc\text{→}tip,\ u\rangle.$$

#**define** *current_arc* $x.A$

16. Busca em profundidade explora um grafo sistematicamente, visitando todos os vértices e verificando para onde estes vértices podem nos conduzir. O vértice ativo que é ponta final do caminho corrente é u.

Há duas possibilidade, o arco $u \to current\_arc$ tenha ponta final em um vértice v que é ativo ou inativo. Se o vértice v é ativo, então v já faz parte do caminho corrente e tentaremos estender o caminho através do arco sucessor de $u \to current\_arc$ na lista de adjacência de u. Caso v seja inativo, marcamos v passa a ser a nova ponta final e 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 ativos, chegou a hora de dar um passo atrás e fazer com que $u \to pred$ volte a ser a ponta final do caminho.

Para tentar encontrar um circuito hamiltoniano no grafo g precederemos da seguinte maneira. O *grau de saída* de um vértice é o número de arcos que têm ponta inicial nele. Inicialmente o programa escolhe um vértice 'destino' t; se a opção `-h` foi usada escolheremos como t um vértice de g que tem grau de saída mínimo. Se encontramos em g um caminho hamiltoniano de s a t, então g é hamiltoniano. De fato, a concatenação do caminho ao arco de t a s é um circuito hamiltoniano.

A idéia básica do algoritmo enumerativo abaixo é a seguinte. Com t fixado tente, para cada arco da forma ts, encontar um caminho hamiltoniano de s a t. É evidente que se todas as buscas por um tal caminho falharem, então o grafo não é hamiltoniano.

17. 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 11 ⟩ +≡
  strcpy($g \to util\_types$, `"ZZZAVIZZZZZZZZ"`);

18. Estrutura geral do algoritmo enumerativo. Seja t um vértice de g. É evidente que se g é hamiltoniano então existe um arco, digamos de t a s, que faz parte desse circuito. Logo, existe um caminho hamiltoniano de s a t em g. Este fato determina o comportamento do trecho de código a seguir: um vértice t e determinado e para cada arco da forma ts o programa procura um caminho hamiltoniano de s a t. Se para algum arco ts um tal caminho é encontrado entõa o grafo é hamiltoniano, caso contrário o programa declara que o grafo dado não é hamiltoniano e pára.

```
⟨ Verifique se g é hamiltoniano 18 ⟩ ≡
  {
    Boolean hamiltoniano = FALSE;
    if (heurística)
      ⟨ Encontre em g um vértice t de grau de saída mínimo 20 ⟩
    else t = g→vertices;
    ⟨ Marque todos vértices como inativos 19 ⟩;
    t→current_arc = t→arcs;
    while (t→current_arc ∧ ¬hamiltoniano)
    {
      s = t→current_arc→tip;
      s→pred = t;
      ⟨ Se existe um caminho hamiltoniano de s a t, hamiltoniano = TRUE 21 ⟩;
      if (¬hamiltoniano) t→current_arc = t→current_arc→next;
    }
    if (hamiltoniano)
    {
      ⟨ Verifique se os current_arc's formam um circuito hamiltoniano 23 ⟩;
```

```
      fprintf(stdout, "O␣grafo␣e␣hamiltoniano.\n");
    }
    else fprintf(stdout, "O␣grafo␣nao␣e␣hamiltoniano.\n");
    fprintf(stdout, "no␣backtrackings:␣%ld.\nno␣nos␣gerados:␣%ld.\n", no_backtrackings, no_nos);
  }
```

Este código é usado no bloco 5.

19. ⟨ Marque todos vértices como inativos 19 ⟩ ≡

```
  {
    register Vertex *v;
    for (v = g->vertices; v < g->vertices + g->n; v++) v->estado = INATIVO;
  }
```

Este código é usado no bloco 18.

20. Ao final deste bloco a variável t contém um vértice em g de grau de saída máximo.

⟨ Encontre em g um vértice t de grau de saída mínimo 20 ⟩ ≡

```
  {
    register Vertex *v;
    register Arc *a;
    unsigned long grau_min = 0;
    t = g->vertices;
    for (a = t->arcs; a; a = a->next) grau_min++;
    for (v = g->vertices + 1; v < g->vertices + g->n; v++)
    {
      unsigned long grau = 0;
      for (a = v->arcs; a; a = a->next) grau++;
      if (grau < grau_min)
      {
        t = v;
        grau_min = grau;
      }
    }
    fprintf(stdout, "Vertice␣origem:␣%s.\nGrau␣de␣saida:␣%lu.\n", t->name, grau_min);
  }
```

Este código é usado no bloco 18.

21. Cada iteração do trecho de código deste bloco começa com um caminho de s a u. Cada iteração tenta estender este caminho a um caminho hamiltoniano que vá de s a t. Assim, por exemplo, se no início de uma iteração tem-se que $u = t$ e o comprimento do caminho é $g \to n - 1$, tem-se que um caminho hamiltoniado de s a t acaba de ser encontrado.

O número de iterações do laço **while** $(\neg tentou\_tudo \wedge \neg hamiltoniano)$ é essencialmente dado pela soma de *no_nos* e *no_backtrackings*. A váriável *no_nos* indica o número de nós gerados na árvore de decisão e *backtrackings* indica o número de arrependimentos feitos pelo programa.

Durante a execução o número de nós gerados da árvore de decisão é contado. A cada 10 Mega nós um ponto ('.') é impresso para indicar que o programa ainda está trabalhando.

⟨ Se existe um caminho hamiltoniano de s a t, *hamiltoniano* = TRUE 21 ⟩ ≡

```
  {
    unsigned long comprimento;    /* número de arcos no caminho corrente */
    Boolean tentou_tudo;     /* indica se todos os candidatos a solução foram testados */
```

```
    u = s;
    u→current_arc = u→arcs;
    u→estado = ATIVO;
    comprimento = 0;
    tentou_tudo = FALSE;
    while (¬tentou_tudo ∧ ¬hamiltoniano)
    {
      if (u ≡ t ∧ comprimento ≡ g→n − 1) hamiltoniano = TRUE;
      else if (u ≡ s ∧ u→current_arc ≡ Λ) {
        u→estado = INATIVO;
        tentou_tudo = TRUE;
      }
      else if (u ≡ t ∨ u→current_arc ≡ Λ) {      /* backtracking */
        no_backtrackings++;
        u→estado = INATIVO;
        comprimento−−;
        u = u→pred;
        u→current_arc = u→current_arc→next;
      }
      else if (u→current_arc→tip→estado ≡ ATIVO)
      {      /* vamos tentar o próximo arco */
        no_nos++;
        u→current_arc = u→current_arc→next;
      }
      else
      {      /* cresce o caminho corrente */
        Vertex *v;
        no_nos++;
        v = u→current_arc→tip;
        v→estado = ATIVO;
        comprimento++;
        v→pred = u;
        if (v ≠ t) v→current_arc = v→arcs;
        u = v;
      }
      if (no_nos % 10000000 ≡ 0)      /* só para ver se o prog. ainda está vivo */
      {
        fprintf(stdout, ".");
        fflush(stdout);
      }
    }
    fprintf(stdout, "\n");
  }
```

Este código é usado no bloco 18.

22.⟨ Variáveis locais do *main* 8 ⟩ +≡

```
  unsigned long no_nos = 0;
  unsigned long no_backtrackings = 0;
```

23. Se o programa encontra um circuito hamiltoniano, então os *current_arc*'s formam este circuito. Neste bloco são contados e marcados todos os vértices que fazem parte do suposto circuito hamiltoniano. Caso os *current_arc*'s não formem um circuito hamiltoniano, o programa para após escrever uma mensagem de erro.

⟨ Verifique se os *current_arc*'s formam um circuito hamiltoniano 23 ⟩ ≡

```
{
  register Arc *a;
  long comprimento;      /* número de arcos percorridos */
  fprintf(stdout, "Circuito␣hamiltoniano:␣");
  fprintf(stdout, "%s", t->name);
  for (a = t->current_arc, comprimento = 1; a->tip ≠ t; a = a->tip->current_arc) {
    fprintf(stdout, ",␣%s", a->tip->name);
    if (a->tip->estado ≡ ATIVO) {
      comprimento++;
      a->tip->estado = MARCADO;
    }
  }
  fprintf(stdout, "\n");
  if (comprimento ≠ g->n) {
    fprintf(stderr,
        "%s:␣os␣current_arc's␣nao␣formam␣um␣circuito␣hamiltoniano!\n""comprimento:␣%ld.\n",
        nome_programa, comprimento);
    return (putz_programa_errado);
  }
}
```

Este código é usado no bloco 18.

24. 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* 24 ⟩ ≡

```
{
  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);
    exit(salva_grafo_erro);    /* Opsss ... */
  }
}
```

Este código é usado no bloco 5.

25.

26. Exercícios

1. Qual o consumo de tempo do programa quando g é um circuito com n vértices (opção `-c`)? A heurística (opção `-h`) reduz o consumo de tempo neste caso?
2. Qual o consumo de tempo do programa quando g é um grafo com n vértices e sem arcos (opção `-e`)? A heurística (opção `-h`) reduz o consumo de tempo neste?
3. Qual o consumo de tempo do programa quando g é um torneio com n vértices (opção `-t`)? A heurística (opção `-h`) reduz o consumo de tempo neste caso?
4. Qual o consumo de tempo do programa quando g é um grafo completo com n vértices (opção `-c`)? A heurística (opção `-h`) reduz o consumo de tempo neste caso?
5. Qual o consumo de tempo assintótico no pior caso deste programa? Apresente um tipo de grafo no qual este consumo de tempo é alcançado.
6. Descreva alguns tipos de grafos para os quais a heurítica do vértice de menor de grau de saída (opção `-h`) faz com que o programa/algoritmo gaste uma quantidade de tempo significativamente menor.
7. Descreva um exemplo em que o tempo gasto pelo programa/algoritmo utilizando a heurística é maior que se não ela não fosse usada.
8. 'Envenene' esta implementação através de mais algumas heurísticas.
9. O `SGB` possui mais geradores de grafos. Acrescente esses geradores a este programa e considere o consumo de tempo do programa tende esses grafos como entrada.
10. Algoritmos de enumeração são muitas vezes escritos de manera recursiva. Quais estruturas de dados que na presente implementação fazem o papel da tal pilha de recursão?
11. Escreva uma função recursiva

$$\textbf{Boolean}(\textbf{Graph}\ *g)\ \{\ \ldots\ \}$$

que devolve `TRUE` se g é hamiltoniano e `FALSE` caso contrário.

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