Localização de instalações

Ana Lucia Marreiros

28 de outubro de 2003

Índice

1. Introdução

Um dos mais importantes aspectos da logística é decidir onde instalar lojas, depósitos, fábricas, servidores http, etc. Essas decisões estratégicas determinam se os produtos fluirão de maneira eficiente através do sistema de distribuição existente.

Em linha gerais, problemas de localização de instalações consistem em determinar locais onde algum recurso deve ser instalado e como estes recursos atenderão as demandas dos centros consumidores ou clientes. Dependendo da aplicação, os locais das possíveis instalações e os clientes podem ser modelados, por exemplo, como vértices em um grafo ou pontos em algum espaço métrico.

Considere, por exemplo, a seguinte versão do problema. Suponha que fábricas de um determinado produto devem ser construídas a fim de atender a demanda de clientes por este produto. Esta situação está ilustrada na figura 1 onde os símbolos $\square$ e $\mathbb{D}$ indicam locais candidatos a receber uma fábrica e os símbolos $\bigcirc$ e $\mathbb{D}$ representam os clientes.

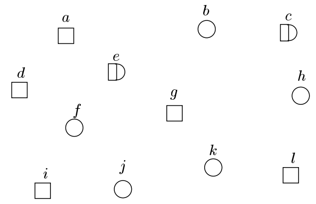

Figura 1: Ilustração do problema da localização de instalações.

Existem certos custos associados ao problema. Para cada local candidato a receber uma fábrica, há um custo de instalar-se uma fábrica nesse local. Além disso, para cada local i candidato a receber uma fábrica e cada cliente j tem-se um custo correspondente a uma fábrica em i satisfazer *toda* a demanda de j.

O problema pede uma lista de locais onde fábricas devem ser instaladas e para cada cliente, qual fábrica deve satisfazer toda sua demanda de maneira que o custo total seja o menor possível (figura 2).

Problemas de localização de instalações incluem uma grande gama de problemas tais como determinar locais onde devem ser construídos postos médicos, depósitos de materiais de risco, caixas eletrônicos, apenas pra enumerar alguns.

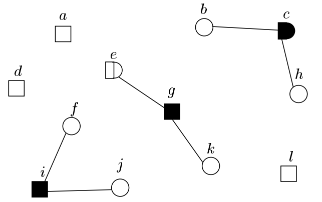

Figura 2: Localização das fábricas e atendimento aos clientes.

2. Problema

Passamos a descrever o problema da localização de instalações mais precisamente. O problema da localização de instalações pode ser modelado através de um grafo bipartido completo (figura 3).

Seja G um grafo bipartido[1] completo com partes F e C, onde F representa um conjunto de locais onde recursos podem ser instalados e C representa centros consumidores ou clientes que têm alguma demanda pelos recursos instalados. Denotamos por E o conjunto das arestas de G.

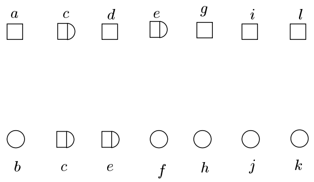

Figura 3: Modelagem da situação da figura 1 como um grafo bipartido.

O custo necessário para instalar recursos é dado por uma função f de F em $\mathbb{Z}_{\geq}$. Assim, $f[i]$ é o custo para instalar o recurso em i. O custo para atender as demandas é dado por uma função c de E em $\mathbb{Z}_{\geq}$. Desta forma, $c[ij]$ é o custo de j utilizar o recurso em i.

Uma **localização** para G é um par (A, L) tal que A é uma parte de F, L é uma parte de $\delta(A)$ tal que $\Gamma_L(A) = C$ (figura 4). (A, L)

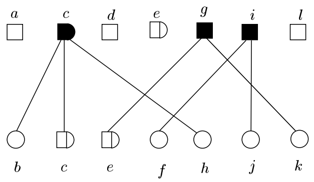

Figura 4: Uma localização para o grafo da figura 3.

O **custo de uma localização** (A, L) é $f(A) + c(L)$, onde $f(A) = \sum_{i \in A} f[i]$ e $c(L) = \sum_{ij \in L} c[ij]$. O custo da localização (A, L) é representado por custo(A, L). custo(A, L)

[1] Talvez fique mais claro se enxergarmos o grafo G como sendo um grafo dirigido contendo todos os arcos da forma ij com i em F e j em C

O **problema da localização de instalações** (*facility location problem*), denotado pela sigla MinLI pode ser descrito como:

Problema MinLI (F, C, f, c): Dados um grafo bipartido completo com partes F e C, um custo $f[i]$ em $\mathbb{Z}_{\geq}$ para cada i em F e um custo $c[ij]$ em $\mathbb{Z}_{\geq}$ para cada ij em E, encontrar uma localização de custo mínimo. MinLI

Como é mostrado a seguir, o problema MinLI é tão difícil quanto o problema da cobertura por conjuntos, denotado por MinCC que por sua vez é equivalente ao problema da transversal mínima.

Seja $\mathcal{S}$ uma coleção de partes de um conjunto finito U. Dizemos que uma parte $\mathcal{T}$ de $\mathcal{S}$ **cobre** U se todo elemento de U pertence a algum conjunto de $\mathcal{T}$. Nesse caso, dizemos também que $\mathcal{T}$ é uma **cobertura** de U. O MinCC consiste no seguinte:

Problema MinCC$(U, \mathcal{S}, f)$: Dados um conjunto finito U, uma coleção $\mathcal{S}$ de partes que cobrem U, e um custo $f[S]$ em $\mathbb{Q}_{\geq}$ para cada S em $\mathcal{S}$, encontrar uma cobertura $\mathcal{T}$ que minimize $f(\mathcal{T})$. MinCC

Uma instância $(U, \mathcal{S}, f)$ de MinCC pode ser transformada em tempo polinomial em uma instância (F, C, f, c) de MinLI. Tome $F = \mathcal{S}$, $C = U$ e seja G o grafo bipartido completo com partes F e C e o conjunto de arestas E. Seja c a função de E em $\mathbb{Z}_{\geq}$, definido da seguinte maneira

$$c[Su] := \begin{cases} 0 & \text{se } u \text{ está em } S, \\ \sum_{S \in \mathcal{S}} f[S] + 1 & \text{se } u \text{ não está em } S. \end{cases}$$

Um exemplo dessa transformação pode ser observado nas figuras 5 e 6.

Pode-se verificar que uma parte $\mathcal{T}$ de $\mathcal{S}$ é uma cobertura de U se e somente se $(\mathcal{T}, L_{\mathcal{T}})$ é uma localização para G, onde $L_{\mathcal{T}} := \{Su : S \in \mathcal{T} \text{ e } u \in S\}$. Ademais, $f[\mathcal{T}]$ é igual a custo$(\mathcal{T}, L)$. Logo, como MinLI, o MinCCé NP-difícil [?].

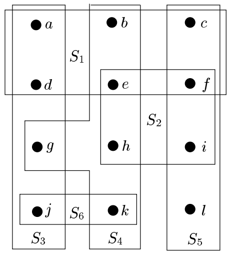

Figura 5: Uma instância do problema de cobertura por conjuntos: $U = \{a, b, c, \ldots, l\}$ e $\mathcal{S} = \{S_1, S_2, ..., S_6\}$.

Todo candidato a solução para MinCC$(U, \mathcal{S}, f)$ corresponde a um candidato a solução para o MinLI(F, C, f, c) de mesmo custo e sendo assim, uma aproximação para MinLI corresponde a uma aproximação para MinCC. Sob a hipótese de P $\neq$ NP, o MinCC tem como limiar de aproximação $\epsilon \log E$, para alguma constante $\epsilon > 0$ [?]. Portanto, o limiar de aproximação para o MinLI é pelo menos o limiar de MinCC.

Uma versão mais tratável do MinLI é aquela onde o custo c é uma função métrica, ou seja, $c[ij] \leq c[ik] + c[kl] + c[lj]$, para todo ij, kl, lj em E (figura 7).

O **problema métrico da localização de instalações**, denotado pela sigla MinMLI pode ser descrito como:

Problema MinMLI(F, C, f, c): Dados um grafo bipartido completo com partes F e C, um custo $f[i]$ em $\mathbb{Z}_{\geq}$ para cada i em F e um custo $c[ij]$ em $\mathbb{Z}_{\geq}$ para cada ij em E tal que c satisfaz a desigualdade triangular, encontrar uma localização de custo mínimo. MinMLI

O valor de uma localização de custo mínimo para o MinMLI(F, C, f, c) é denotado por opt(F, C, f, c). Esta é a versão do problema que é tratada no resto deste capítulo.

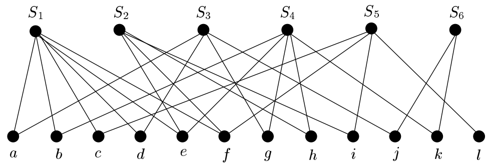

Figura 6: Representação da instância da figura 5 como uma instância do problema de localização de instalações. As arestas ausentes na figura têm custo 'infinito'.

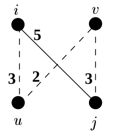

Figura 7: Ilustração da desigualdade triangular.

3. 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).

4. Como usar o programa

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

`meu_prompt> localiza ⟨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`).

`-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.

`-f`⟨`número`⟩ O grafo bipartido completo gerado terá f vértices na primeira partição, onde f é o ⟨número⟩ dado.

`-c`⟨`número`⟩ O grafo bipartido completo gerado terá c vértices na segunda partição, onde c é o ⟨número⟩ dado.

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

É evidente que só faz sentido usar apenas uma entre as opções `-g`, `-r`. Também não faz sentido usar a opção `-g` junto com as opções `-f` e `-c`. 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* 4 ⟩ ≡

```
void mostre_uso( )
{
  fprintf (stdout,
  "Uso:␣%s␣[-r|-g<nome>]␣[-s<num>]␣[-f<num>]␣""[-c<num>]␣[-o<nome>]\n"
  "␣␣␣␣␣-r␣␣grafo␣pseudo-aleatorio␣com␣n␣vertices␣e␣m␣arestas\n"
  "␣␣␣␣␣-g<nome>␣trabalha␣com␣o␣grafo␣do␣arquivo␣<nome>\n"
  "␣␣␣␣␣-s<num>␣␣para␣o␣gerar␣numeros␣pseudo-aleatorios\n"
  "␣␣␣␣␣-f<num>␣numero␣de␣vertices␣na␣1a.␣particao\n"
  "␣␣␣␣␣-c<num>␣numero␣de␣vertices␣na␣2a.␣particao\n"
  "␣␣␣␣␣-o<nome>␣grafo␣de␣trabalho␣sera␣salvo␣no␣arquivo␣<nome>\n"
  , nome_programa);
  exit(0);
}
```

Este código é usado no bloco 8.

5. 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.

6. 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 7 ⟩
  ⟨ Declaração de funções 8 ⟩
  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 9 ⟩;
    ⟨ Examine as opções na linha de comando 10 ⟩;
    ⟨ Crie o grafo g dado 12 ⟩;
    ⟨ Encontre uma localização para g 19 ⟩;
    if (salva_arquivo) ⟨ Salve o grafo g no arquivo salva_arquivo 21 ⟩;
    gb_recycle(g);    /* devolve a memória usada por g */
    return 0;    /* termino normal */
  }
```

7. 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 7 ⟩ ≡
  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 14.
Este código é usado no bloco 6.

8. 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 8 ⟩ ≡
  ⟨ Função mostre_uso 4 ⟩
```

Este código é usado no bloco 6.

9. 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* 9 ⟩ ≡

```
unsigned long n = 20;   /* número de vértices */
unsigned long m = 100;   /* número de arcos */
unsigned long f = 20;   /* número de vértices na 1a. partição */
unsigned long c = 20;   /* número de vértices na 2a. partição */
long directed = 0;   /* o grafo gerado será simétrico */
long seed = 0;   /* semente para o gerador pseudo-aleatório */
Boolean aleatório = FALSE;   /* gerar um grafo pseudo-aleatório */
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 11.
Este código é usado no bloco 6.

10. 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 -f0 -m1000`; note que todo circuito com 10 vértices tem 20 arcos.

⟨ Examine as opções na linha de comando 10 ⟩ ≡

```
nome_programa = argv[0];
if (argc ≡ 1) mostre_uso();
while (−−argc) {
  if (strcmp(argv[argc], "-r") ≡ 0 ∧ ¬escolheu) escolheu = aleatório = TRUE;
  else if (strncmp(argv[argc], "-g", 2) ≡ 0 ∧ ¬escolheu) {
    escolheu = TRUE;
    nome_arquivo = argv[argc] + 2;
  }
  else if (sscanf(argv[argc], "-f%lu", &f) ≡ 1) ;
  else if (sscanf(argv[argc], "-c%lu", &c) ≡ 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|-g<nome>]␣[-s<semente>]␣[-f<num>]␣[-c<num>]␣[-o<nome>]\n",
        nome_programa);
    exit(opção_invalida);
  }
}
if (¬escolheu) aleatório = FALSE;   /* opção default */
```

Este código é usado no bloco 6.

11. 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* 9 ⟩ +≡

```
Graph *g = Λ;   /* grafo bipartido de trabalho */
```

12. 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 12 ⟩ ≡

```
if (aleatório) g = random_graph(n, m, 0, 0, directed, Λ, Λ, 1, 1, seed);
else if (nome_arquivo) g = restore_graph(nome_arquivo);
else {
  g = bi_complete(f, c, directed);
  strcpy(g→util_types, "ZZZZZZZZZZZZZZ");
}
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);
```

Este código é usado no bloco 6.

13. 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.

14. 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 7 ⟩ +≡
  enum {
    `INATIVO` = 0, `ATIVO` = 1, `MARCADO` = 2
  };

15. 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 $w{\to}pred \equiv v$. 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

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

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

16. 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, $w{\to}pred = v$. O arco de v a w também é representado diretamente na estrutura pelo campo *current_arc* do vértice v. Desta forma, tem-se que $v{\to}current\_arc{\to}tip = w$. 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

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

#define *current_arc* $x.A$

17. 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 procederemos 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.

18. 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).

19. Estrutura geral do algoritmo. 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.

⟨ Encontre uma localização para g 19 ⟩ ≡
  {
  }

Este código é usado no bloco 6.

20.

21. 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* 21 ⟩ ≡

```
{
  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 6.

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