Grafos

Juliana Kaizer Vizzotto

Universidade Federal de Santa Maria

Estrutura de Dados

Roteiro

- Introdução
- Tipos de Grafos
- Implementação de Grafos

Introdução

- Grafos: representação abstrata que descreve a organização de sistemas de transporte, circuitos elétricos, interações humanas, redes de telecomunicações.

- Vamos começar estudando possíveis maneiras de representar grafos computacionalmente e também alguns algoritmos mais elementares para percurso em grafos.

Grafos

- Um grafo $G = (V, E)$ é definido por um conjuntde *vértices* (ou nodos), V, e um conjunto de *arcos* (ou ligações), E, constituído de um conjunto de pares ordenados (ou não) de vértices de V.
- Por exemplo, modelando uma rede de estradas, os vértices podem representar as cidades e os arcos as conexões entre as cidades.
- Códigos fontes de programas também podem ser modelados com grafos: os vértices podem representar linhas de código, tal que os arcos conectando linhas x e y significam que y deve ser executado depois de x.

Tipos de Grafos

- *Direcionados* e *Não-Direcionados*;
- *Com pesos* e *Sem pesos*;
- *Cíclicos* e *Acíclicos*;
- *Com Labels* e *Sem Labels*;

Representando Grafos

- Existem diversas maneira de representar grafos. Vamos discutir 4 maneiras possíveis. Assuma que o grafo $G = (V, E)$ contém n vértices e m arcos.
- Matriz de Adjacência: Podemos representar G usando uma matriz $M_{n \times n}$, talque que o elemento $M[i, j] = 1$, se (i, j) é um arco do grafo G, e $M[i, j] = 0$, caso contrário.
- Essa representação, embora simples, pode em algumas aplicações usar espaço excessivo. Principalmente em grafos com muitos vértices e poucos arcos.

Representando Grafos

- Listas de Adjacência em Listas: podemos representar grafos esparsos utilizando listas ligadas para armazenar os vizinhos adjacentes de cada vértice.
- Listas de Adjacência requerem ponteiros e é um pouquinho mais complicado saber quando o par (i, j) é um arco de G, pois temos que buscar na lista apropriada para encontrar o arco.
- Entretanto, é bastante simples trabalhar com algoritmos de grafos, os quais não precisam responder tais perguntas.
- Tipicamente, podemos varrer o grafo passando por todos os arcos em uma passada via busca em largura e busca em profundidade.

Representando Grafos

- Listas de Adjacência em Matrizes: Listas de adjacência também podem ser armazenadas em matrizes, então eliminando a necessidade de ponteiros.
- Podemos representar uma lista em uma linha da matriz, guardando um contador k de quantos elementos existem e guardando eles nos k primeiros elementos do vetor (linha).
- Então, podemos visitar os elementos sucessivos como numa lista, incrementando o índice, sem a necessidade de trabalhar com ponteiros.
- Parece que essa estrutura combina as piores propriedades de matrizes de adjacência (muito espaço) com as piores propriedades de listas de adjacência (necessidade de busca por arcos).

Representando Grafos

- Entretanto, ela é a estrutura de dados mais simples de programar, particularmente para grafos estáticos, i.e., que não mudam após sua construção.
- E o problema de espaço pode, em princípio, ser eliminado através da alocação de linhas para cada vértice dinamicamente, fazendo com que as linhas tenham o tamanho exato.
- Vamos usar essa representação nos nossos exemplos!

Implementando Grafos: listas de adjacência em matrizes

- Para cada grafo, guardamos um contados do número de vértices, e atribuímos um número a cada um, de 1 `nvertices`.
- Vamos representar os arcos em um vetor `MAXV x MAXDEGREE`, tal que cada vértice pode ter no máximo `MAXDEGREE` adjacentes.

Implementando Grafos: listas de adjacência em matrizes

```
#define MAXV 100
#define MAXDEGREE 50
typedef struct graph{
    int edges[MAXV+1][MAXDEGREE]; //info de adjacencia
    int degree[MAXV+1]; //degree de cada vertice
    int nvertices;
    int nedges;
}
```

Lendo um grafo de um arquivo

Um formato típico consiste de uma linha inicial contendo o número de vértices e arcos, seguidos por uma listas de arcos como um par de vértices por linha.

```
read_graph(graph *g,int directed){
    int i, m, x, y;
    initialize_graph(g);
    scanf("%d %d",&g->nvertices),&m);
    for (i=1;i<=m;i++){
        scanf("%d d%",&x,&y);
        insert_edge(g,x,y,directed);
    }
}
```

Inicializando o Grafo

```
initialize_graph(graph *g){
    int i;
    g -> nvertices = 0;
    g -> nedges = 0;
    for (i=1;i<=MAXV;i++) g->degree[i]=0;
}
```

Inserindo Arcos

```
insert_edge(graph *g,int x,int y,int directed){
  if(g->degree[x] > MAXDEGREE)
     printf("Excedeu o nível máximo\n");
  g->edges[x][g->degree[x]] = y;
  g->degree[x] ++;
  if (directed == FALSE)
      insert_edge(g,y,x,TRUE);
  else
      g->nedges ++;
}
```

Imprimindo o Grafo

```
print_graph(graph *g){
    int i,j;
    for (i=1;i<=g->nvertices;i++){
      printf("%d: ",i);
      for (j=0;j<- g->degree[i];j++)
         printf(" %d",g->edges[i][j]);
      printf("\n");
    }
}
```

Percurso em Grafos

- A operação básica na maioria dos algoritmos de grafos é sistematicamente percorrer o grafo.
- Queremos visitar cada vértice e cada arco exatamente uma única vez em algum ordem pré-definida.
- Existem dois algoritmos mais importantes de percurso: em largura (*breadth-first* - BFS) e em profundidade (*depth-first* - DFS).
- Para alguns problemas não faz diferença qual dos métodos usamos, mas em outros casos a diferença é crucial.
- Ambos procedimentos compartilham uma ideia fundamental, é necessário marcar os vértices que já foram visitados antes, assim não vamos explorá-los denovo.
- BFS e DFS são diferentes na ordem em que exploramos os vértices.

Largura

- BFS é apropriado se não importa a ordem na qual visitamos os vértices do grafo.
- Ou também quando estamos interessados em caminhos mais curtos em grafos sem peso.
- Nossa implementação de BFS usa dois vetores booleanos para manter o conhecimento sobre cada vértice no grafo.
- Um vértice é descoberto a primeira vez que ele é visitado.
- Um vértice é considerado processado depois que percorremos todos os vértices que partem dele.
- Assim, um vértice passa primeiramente a descoberto e depois processado.

Largura

- Uma vez que um vértice é descoberto, ele é colocado em uma fila.
- Como os vértices são processados com a estratégia FIFO, os vértices mais velhos são expandidos primeiro (aqueles mais perto da raiz).

Percurso em Largura

```
bfs(graph *g,int start){
  queue q; int i,v;
  enqueue(&q,start); discovered[start] = TRUE;
  while (empty(&q) == FALSE){
     v =dequeue(&q);
     process_vertex(v);
     processed[v]=TRUE;
     for (i=0;i<g->degree[v];i++)
              if (discovered[g->edges[v][i]] == FALSE){
                 enqueue(&q,g->edges[v][i]);
                 discovered[g->edges[v][i]] = TRUE;
                 parent[g->edges[v][i]] = v;}
              if (processed[g->edges[v][i] == FALSE);
                  process_edge(v,g->edges[v][i]);}}}
```

Largura

- O comportamento exato do bfs depende das funções `process_vertex()` e `process_edge()`.
- Através dessas funções, podemos facilmente customizar o que o percurso faz, já que ele oficialmente visita cada vértice e cada arco.
- Por exemplo:

Largura

```
process_vertex(int v){
  printf("Vertice Processado %d \n",v);
}
process_edge(int x,int y){
  printf("Arco Processado (%d,%d) \n",x,y);
}
```

Largura

- A função `valid_edge`, possibilita que ignoremos a existência de um certo número de arcos no grafo.

- Fazendo com que `valid_edge` retorne `TRUE` para todos os arcos resulta em um BFS completo.

Procurando Caminhos

- O vetor `parent` no BFS é útil para encontrar caminhos interessantes no grafo.

Profundidade

- DFS usa essencialmente a mesma ideia do *backtracking*.
- Ambos envolvem busca exaustiva de todas as possibilidades avançando sempre que possível, e voltando sempre que não existe mais possibilidade inexplorada adiante.
- Ambos são mais facilmente entendidos como algoritmos recursivos.
- DFS pode ser pensado como um BFS com uma pilha ao invés de uma fila.
- A *beleza* de implementar *recursivamente* elimina a necessidade de manter a pilha explicitamente.

Profundidade

- dfs.c

Profundidade: encontrando ciclos

- DFS de um grafo sem direção particiona os arcos em duas classes:arcos de árvore e arcos de volta.
- Os arcos de árvore são aqueles codificados na relação `parent`, os arcos que descobrem novos arcos.
- Arcos de volta são aqueles cujos final é um ancestral do vértice sendo expandido.
- Essa é uma propriedade da busca DFS.
- No DFS, todos os vértices alcançados a partir de um dado vértice v são expandidos antes que terminemos o percurso a partir de v.

Profundidade: encontrando ciclos

Achando Ciclos

```
process_edge(int x, int y){
  if (parent[x] != y){ //encontrou ciclo
     printf("Ciclo de %d para %d:",y,x);
     finf_path(y,x,parent);
     finished = TRUE;
  }
}
```

`finished` termina depois de encontrar o primeiro ciclo.

Componentes Conectados

- Um componente conectado de um grafo sem direção é um conjunto maximal de vértices tal que existe um caminho entre cada par de vértices.
- Esses são pedaços separados do grafo, tal que não existe conexão entre os pedaços.
- Um grande número de problemas complicados com grafos se reduzem a encontrar ou contar *componentes conectados*.
- Componentes conectados podem facilmente ser encontrados usando dfs ou bfs, já que a ordem não importa.

Componentes Conectados

- Basicamente, fizemos a busca a partir do primeiro vértice.
- Então, repetimos a busca a partir de qualquer vértice não-descoberto (se existe) para definir o próximo componente, e assim até que todos os vértices sejam encontrados.

Componentes Conectados

- connected-comp.c

Exercício 1. PC/UVa IDs: 110901/10004

- Bicoloring:The four-color theorem states that every planar map can be colored using only four colors in such a way that no region is colored using the same color as a neighbor. After being open for over 100 years, the theorem was proven in 1976 with the assistance of a computer. Here you are asked to solve a simpler problem. Decide whether a given connected graph can be bicolored, i.e., can the vertices be painted red and black such that no two adjacent vertices have the same color. To simplify the problem, you can assume the graph will be connected, undirected, and not contain self-loops (i.e., edges from a vertex to itself).

Exercício 1. PC/UVa IDs: 110901/10004

- Bicoloring:The four-color theorem states that every planar map can be colored using only four colors in such a way that no region is colored using the same color as a neighbor. After being open for over 100 years, the theorem was proven in 1976 with the assistance of a computer. Here you are asked to solve a simpler problem. Decide whether a given connected graph can be bicolored, i.e., can the vertices be painted red and black such that no two adjacent vertices have the same color. To simplify the problem, you can assume the graph will be connected, undirected, and not contain self-loops (i.e., edges from a vertex to itself).

Exercício 1. PC/UVa IDs: 110901/10004

- Input: The input consists of several test cases. Each test case starts with a line containing the number of vertices n, where 1 ¡ n ¡ 200. Each vertex is labeled by a number from 0 to n−1. The second line contains the number of edges l. After this, l lines follow, each containing two vertex numbers specifying an edge. An input with n = 0 marks the end of the input and is not to be processed.

Exercício 1. PC/UVa IDs: 110901/10004

- Output: Decide whether the input graph can be bicolored, and print the result as shown below.

Sample Input	Sample output
3	NOT BICOLORABLE.
3	
0 1	
1 2	
2 0	
0	