MAC0328 - Algoritmos em Grafos - Lista 1

Pedro Paulo Vezzá Campos - 7538743

10 de Abril de 2012

Questão 1

Sejam `d[`w`]`, `f[`w`]` e `parnt[`w`]` respectivamente os tempos de descoberta, finalização e o vértice pai na arborescência DFS do vértice w.

A partir do momento que o algoritmo de busca em profundidade descobre o vértice u temos algumas possibilidades:

O vértice v ainda não foi descoberto.

1. Caso o algoritmo descubra v através de $u \to v$ temos que `d[`u`]` $<$ `d[`v`]` e `parnt[`v`]` $= u$. Ainda, vale `f[`v`]` $<$ `f[`u`]` pela definição de DFS. Portanto $u \to v$ é arco de arborescência. Automaticamente o arco $v \to u$ será de retorno pela definição de arco de retorno (`d[`u`]` $<$ `d[`v`]` $<$ `f[`v`]` $<$ `f[`u`]`).
2. Caso o algortimo descubra v por outro caminho que não $u \to v$ temos novamente que `d[`u`]` $<$ `d[`v`]` e `f[`v`]` $<$ `f[`u`]` pois v é um descendente de u. Neste caso, no entanto, `parnt[`v`]` $\neq u$ pois não foi utilizado o arco $u \to v$ para descobrir v. Isso denota que $u \to v$ é um arco descendente. Novamente, $v \to u$ será arco de retorno pela definição de arco de retorno.

O vértice v já foi descoberto mas ainda não foi fechado.

1. Caso o vértice u tenha sido descoberto através do arco $v \to$ u estamos em um caso análogo ao 1.1. Vale que `d[`v`]` $<$ `d[`u`]` e `parnt[`u`]` $= v$ e pela definição de DFS vale `f[`u`]` $<$ `f[`v`]`. Com isso concluímos que $v \to u$ é da arborescência e que $u \to v$ é de retorno.
2. Caso o algortimo descubra v por outro caminho que não $v \to u$ estamos em um caso análogo ao 1.2. Vale novamente que `d[`v`]` $<$ `d[`u`]` já que u é descendente de v. Mas `parnt[`v`]` $\neq v$ pois não utilizamos $v \to u$ para descobrir u. Concluímos assim que $v \to u$ é um arco descendente. Novamente, $u \to v$ será arco de retorno pela definição de arco de retorno.

O vértice v já foi descoberto e fechado. (Impossível!)

Isso implicaria que todos os vértices atingíveis a partir de v já teriam sido descobertos e fechados, inclusive u, o que contraria a suposição que o algoritmo acabou de descobrir u. Isso elimina a possibilidade que $u \to v$ seja um arco cruzado pois não é possível que `d[`v`]` $<$ `f[`v`]` $<$ `d[`u`]` $<$ `f[`u`]`. Analogamente, também não é possível que $v \to u$ seja arco cruzado pois isso implicaria que `d[`u`]` $<$ `f[`u`]` $<$ `d[`v`]` $<$ `f[`v`]` o que significa que u e todos seus atingíveis já foram fechados, o que contraria a suposição que o algoritmo ainda está percorrendo u. Contradição.

Questão 2

```
/**
all_articulations recebe como parametro um grafo inicializado e populado G
e imprime na saida padrao todas as articulacoes do grafo, uma por linha.
*/
void all_articulations (Graph G) {
  Vertex v;
  cnt = 0;
  for (v = 0; v < G->V; v++)
    pre[v] = -1;
  for (v = 0; v < G->V; v++)
    if (pre[v] == -1) {
      parnt[v] = v;
      articulationR(G, v);
    }
}

void articulationR (Graph G, Vertex v) {
  link p; Vertex w;
  int adj = 0, articulation = 0;
  pre[v] = cnt++;
  low[v] = pre[v];
  for (p=G->adj[v];p!=NULL;p=p->next)
    if (pre[w=p->w] == -1) {
      adj++;
      parnt[w] = v;
      articulationR(G, w);

      if (low[v] > low[w]) low[v]=low[w];
      if (parnt[v] != v && low[w] >= pre[v]) /* Modificacao 1 */
        articulation = 1;
    } else if (w!=parnt[v] && low[v]>pre[w])
      low[v] = pre[w];

  if(articulation || parnt[v] == v && adj > 1) /* Modificacao 2 */
    printf("%d\n", v);
}
```

Os vetores `pre`, `low` e `parnt` continuam tendo a mesma definição utilizada no algoritmo de busca de pontes `all_bridges`.

Seja v um ponto de articulação. Como a sua remoção implica que no aumento do número de componentes, temos que todo caminho que ligue um descendente de v a algum vértice descoberto antes de v deve obrigatoriamente passar por v. Mais especificamente, nenhum filho w de v (`parnt[`w`]` $= v$) terá como acessar algum ancestral de v sem passar por ele. Consequentemente, teremos que o menor número de preordem atingível por w será limitado inferiormente pelo número de preordem de v (`low[`w`]` $\geq$ `pre[`v`]`).

As raizes da floresta DFS são tratadas como casos particulares: Uma raiz só será será uma articulação se ela tiver pelo menos dois nós filhos na arborescência DFS. Caso tenha menos que isso, a remoção da raiz não implica no aumento de componentes, não caracterizando uma articulação. Por outro lado, ter dois ou mais filhos na arborecência implica pela definição de DFS que cada um deles não consegue atingir o outro sem atravessar a raiz para isso.

No algoritmo `all_articulations` as modificações realizadas buscam capturar as duas possibilidades de descoberta de articulação descritas nos parágrafos anteriores. Para isso, caso a DFS encontre um novo filho w do vértice atual v na arborescência, varre-o recursivamente e checa em seguida se vale que `low[`w`]` $\geq$ `pre[`v`]`. Em caso positivo, atualiza a *flag* `articulation` para verdadeiro, indicando que foi encontrada uma nova articulação (Modificação 1).

Ao final da varredura dos adjacentes é verificado se a *flag* `articulation` está marcada como verdadeira ou se é raiz com mais de um filho. Em ambos os casos, imprime o vértice atual indicando que uma nova articulação foi encontrada.

Questão 3

```
/**
GRAPHremoveMultiV recebe como parametros:
 - Um Graph G inicializado e propriamente populado
 - Um vetor removed de flags com tamanho pelo menos G->V
 - Um vetor map Vertex com tamanho pelo menos G->V

Pre-condicoes:
 - G esta inicializado e propriamente populado
 - Para todo vertice v em G vale que removed[v] = 1 caso o vertice v
tenha sido removido e removed[v] = 0 caso contrario.

Pos-condicoes:
 - G nao sofre alteracoes
 - O algoritmo retorna um novo grafo de tamanho G->V - x, sendo x o
numero de vertices removidos
 - O grafo retornado possui todos os vertices nao removidos com nomes
possivelmente trocados
e todas as arestas que nao tinham ponta em um vertice removido.
 - Para todo vertice v de G temos que map[v] = w sendo w o novo nome
do vertice no novo grafo retornado caso v nao tenha sido removido ou
map[v] = -1 caso contrario.

Observacoes:
 - Funcoes auxiliares GRAPHinit e DIGRAPHinsertA sao as mesmas das
notas de aula.
*/
Graph GRAPHremoveMultiV(Graph G, int removed[], Vertex map[]){
  Vertex i, count = 0;
  link p;
  Graph G2;
  for(i = 0; i < G->V; i++)
    map[i] = removed[i] ? -1 : count++;

  G2 = GRAPHinit(count);

  for(i = 0; i < G->V; i++) {
    if(removed[i])
      continue;
    for(p = G->adj[i]; p != NULL; p = p->next)
      if(!removed[p->w])
        DIGRAPHinsertA(G2, map[i], map[p->w]);
  }
```

```
  return G2;
}
```