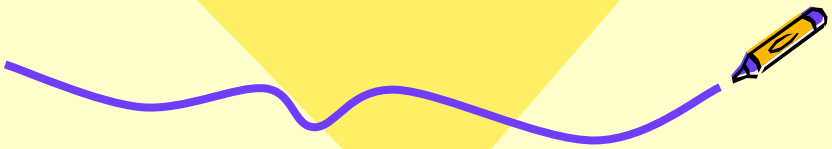
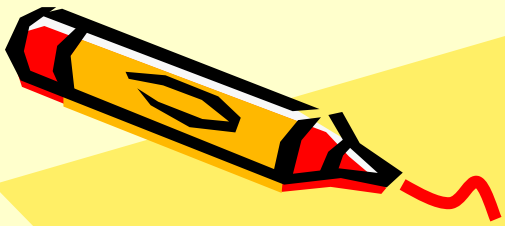
Laboratorio di Algoritmi e Strutture Dati

Aniello Murano
http://people.na.infn.it/~murano/





Operazioni su Grafi: Inserimento e Cancellazione di un Nodo

Possibili scenari

- Bisogna distinguere i seguenti casi:
  - Rappresentazione (matrice di adiacenza/lista di adiacenza)
  - Grafo orientato/non orientato

Matrice di adiacenza

- Per aggiungere o eliminare un vertice in un grafo rappresentato con matrici di adiacenza, abbiamo bisogno di definire la matrice di rappresentazione dinamicamente.
- Una matrice definita dinamicamente può essere passata alle funzioni progettate per lavorare con matrici di dimensioni diverse.

Allocazione di memoria dinamica di una matrice di adiacenza

- Sia G un grafo con n vertici e M[n,n] la matrice di adiacenza corrispondente. Consideriamo le seguenti possibilità di allocazione di memoria dinamica per la matrice M:
  - **Scelta 1.1:** M è un vettore di n x n elementi:

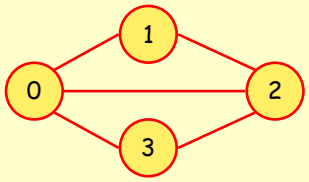
```
int n,*M;
M = (int *) malloc(sizeof(int)*n*n);
```

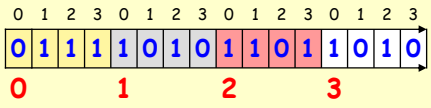
  - **Scelta 1.2:** M è un vettore di puntatori:

```
int n, i,**M;
M = (int **) malloc(n*sizeof(int*));
for (i=0; i<n;++1)
  M[i] = (int *) malloc(n*sizeof(int));
```

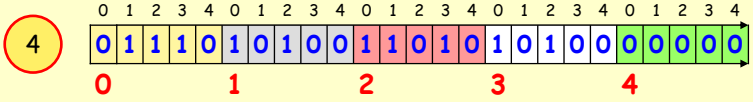

Scelta 1.1: Inserimento

- La matrice è un array di n blocchi di n valori. Per esempio il secondo blocco descrive gli archi uscenti dal nodo 1:





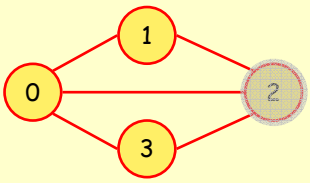
- Utilizzando questa tecnica, l'inserimento di un nodo consiste in una reallocazione di memoria (per aggiungere n+1 celle nel vettore) e in uno spostamento opportuno degli elementi:
  - Il blocco relativo al nodo 0 resta al suo posto, il blocco relativo al nodo 1 viene spostato di una posizione verso destra... e così via, fino al blocco n.
  - Infine si memorizzano i nuovi archi.



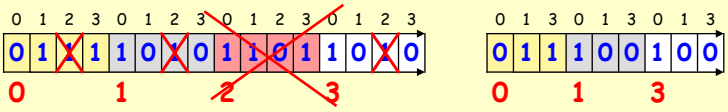
Scelta 1.1: Cancellazione

- Utilizzando la scelta 1.1, la cancellazione di un nodo comporta lo spostamento opportuno degli elementi nel vettore e poi in una reallocazione del vettore nel seguente modo:
- Supponendo di voler cancellare il nodo k-esimo, in un vettore di n blocchi, il codice è il seguente:

```
for(i=0;i<n*n;i++) {
   M[pos]=M[i];
   if (( (i%n) !=k) || ( (i/n) !=k)) pos++; }
```

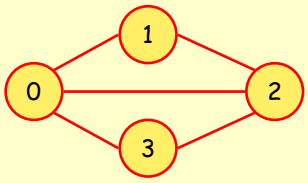


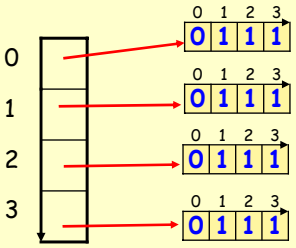
- Per esempio, volendo cancellare il nodo 2 dall'esempio precedente, il risultato è il seguente:



Scelta 1.2

- La matrice è un vettore di puntatori:

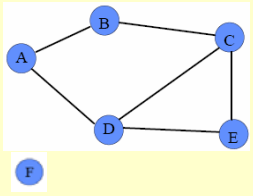


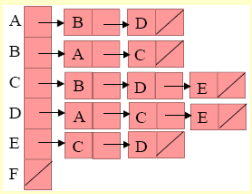


- Utilizzando questa tecnica, l'inserimento di un nodo consiste in una reallocazione di memoria (aggiunta di una cella a puntatore), una allocazione di memoria per M[n] e la memorizzazione dei nuovi archi.
- La cancellazione invece di un nodo **i** consiste nella eliminazione del puntatore **M[i]** shiftando i puntatori in M che seguono **i**, nella deallocazione di memoria per gli archi incidenti a **i**, e nella deallocazione della memoria per il vettore ***M[i]**.

Grafo con liste di adiacenza

```
typedef struct graph {
   int     nv;   /* numero di vertici del grafo */
   edge **adj;  /* vettore con le liste delle adiacenze */ } graph ;
```



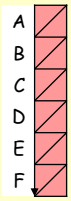



```
typedef struct edge  {
   int key;
   struct edge *next; } edge;
```


Creazione di un grafo

La seguente funzione crea e restituisce un puntatore ad una struttura dati grafo che conserva il numero dei vertici e definisce n liste di adiacenza vuote.

```
graph *g_empty(int n)                    {
  graph *G;  int i;
  G = (graph*)malloc(sizeof(graph));
  if (G==NULL) printf("ERRORE: impossibile allocare memoria per il grafo\n");
  else {
     G->adj = (edge**)malloc(n*sizeof(edge*));
     if ((G->adj==NULL) && (n>0)) {
          printf("ERRORE: impossibile allocare memoria per la lista del grafo\n");
          free(G);
          G=NULL;}
     else {
          G->nv = n;
          for (i=0; i<n; i++)
              G->adj[i]=NULL;}}
  return(G);                             }
```



Inserimento di un nodo

```
graph *g_insert(G)                    {
   edge *e;
   if (G==NULL) return graph *g_empty(1);
   e = realloc(G->adj, (G->nv+1) *sizeof(edge*));
   if ((e ==NULL) {
            printf("ERRORE: impossibile reallocare memoria \n");}
   else {
            G->adj=e;
            G->adj[G->nv]=NULL;
            G->nv = G->nv+1;}
   return(G);                         }
```

- Si ricorda che con realloc se lo spazio di memoria non può essere reallocato, il puntatore oggetto della realloc (in questo caso G->adj) rimane invariato e viene restituito Null.

Cancellazione di un nodo

- Le operazioni da fare sono reallocazione di memoria per il grafo (per il vettore di liste), cancellazione di una intera lista eliminando ogni suo elemento e poi facendo il free, ed infine eliminando tutti gli archi che vengono cancellati con il nodo.
- Esercitazione: implementare il codice corrispondente.

Cancellazione di un grafo

- La seguente funzione libera la memoria occupata da un grafo.

```
void g_free(graph *G) {
   int i; edge *e, *enext;
   if (G!=NULL)        {
     if (G->nv > 0)                          {
              for (i=0; i<G->nv; i++)                {
                       e=G->adj[i];
                       while (e!=NULL) {
                                enext=e->next;
                                free(e);
                                e=enext;}          }
              free(G->adj);                }
     free(G);            }
   return;
}
```


Attraversamento in profondità (DFS)

- L'attraversamento in profondità (deep-first search, DFS) di un grafo non orientato consiste nel visitare ogni nodo del grafo secondo un ordine compatibile con quanto qui di seguito speci-ficato:
- Il prossimo nodo da visitare è connesso con un arco al nodo più recentemente visitato che abbia archi che lo connettano a nodi non ancora visitati.
- L'attraversamento DFS, fra le altre cose, permette l'individuazione delle componenti connesse di un grafo.
- Intuitivamente, la visita in profondità si può schematizzare nel modo seguente:

visita_profondità(*G*)
1. Passo base
   se *G* ==NULL esci;
2. Passo di induzione
   visita il nodo *G* se non è stato visitato
   per ogni nodo adiacente *G->adj*
      visita_profondità(*G->adj*);

Idea di implementazione

- Implementiamo la visita DFS tramite una semplice funzione ricorsiva:
- La procedura implementata usa un array di appoggio aux per memorizzare quando un vertice è stato già incontrato. Inoltre, la funzione principale chiama al suo interno un'altra procedura: dfs1
- Quando dfs1 è richiamata si entra in una nuova componente connessa.
- dfs1 richiamerà se stessa ricorsivamente fino a quando tutta la componente è stata visitata.

Codice per DFS

- Di seguito mostriamo il codice per dfs e dfs1:

```
void dfs(struct graph *g) {
        int i, *aux = calloc(g->nv,sizeof(int));
        if(!aux) {printf("Errore di Allocazione\n");}
        else {
                for(i = 0; i < g->nv; i++)
                if(!aux[i]) {printf("\n%d,",i); dfs1(g,i,aux);}
        free(aux);}

void dfs1(struct graph *g, int i, int *aux) {
        edge *e;
        aux[i] = 1;
        for(e = g->adj[i]; e; e = e->next)
                if(!aux[e->key ]) { printf("%d,",e->v); dfs1(g,e->key,aux);}
        }
```

- Poiché dfs1 contiene un ciclo sulla lista di adiacenza del nodo con cui è richiamata, ogni arco viene esaminato in totale due volte, mentre la lista di adiacenza di ogni vertice è scandita una volta sola.
- La visita DFS con liste di adiacenza richiede O(|V |+|E|).

Attraversamento in Ampiezza(BFS)

- L'attraversamento in ampiezza (breadth-first search, BFS) è un modo alternativo al DFS per visitare ogni nodo di un grafo non orientato.
- Il prossimo nodo da visitare lo si sceglie fra quelli che siano connessi al nodo visitato meno recentemente che abbia archi che lo connettano a nodi non ancora visitati.
- Vediamone un'implementazione non ricorsiva che memorizza in una coda i nodi connessi al nodo appena visitato.
- Sostituendo la coda con uno stack si ottiene una (leggera variante della) visita DFS.

Il codice per BFS

```
void bfs(struct graph *g) {
        int i, *aux = calloc(g->V,sizeof(int));
        if(!aux) { printf("Errore di Allocazione\n"); }
        else {
                for(i = 0; i < g->nv; i++)
                        if(!aux[i]) { printf("\n%d,",i+1); bfs1(g,i,aux); }
                free(aux); }

void bfs1(struct graph *g, int i, int *aux) {
        edge *e;
        intqueue *q = createqueue();
        enqueue(q,i);
        while(!emptyq(q)) {
            i = dequeue(q);
            aux[i] = 1;
            for(e = g->adj[i]; e; e = e->next)
                if(!aux[e->key]) {
                    enqueue(q,e->key); printf("%d,",e->key); aux[e->key] = 1; }}
        destroyqueue(q);}
```

Idea di implementazione di BFS

- La procedura implementata per la visita BFS usa un array di appoggio aux per memorizzare quando un vertice è già stato incontrato.
- Quando bfs1 è richiamata da bfs si entra in una nuova componente connessa.
- bfs1 usa una coda per memorizzare da quali vertici riprendere la visita quando la lista di adiacenza del vertice corrente è stata tutta esplorata.
- Ogni lista è visitata una volta, e ogni arco due volte.
- La visita BFS con liste di adiacenza richiede $O(|V|+|E|)$.