Rappresentazione di Grafi (non orientati)

Vi sono molti modi di rappresentare grafi non orientati.

Quando si deve scegliere come implementare grafi (orientati o meno) in una certa applicazione, bisogna considerare gli eventuali vincoli — impliciti o espliciti — e le possibili operazioni che caratterizzano la classe di grafi adatta all'applicazione.

Consideriamo due rappresentazioni alternative per grafi non orientati generici:
– con *matrice* d'adiacenza.
– con *liste* d'adiacenza.

Matrice d'adiacenza.

Consideriamo un grafo non orientato $G = (V, E)$. Numeriamo i vertici di V: Indichiamo con v_i il vertice di V la cui etichetta è $i \in \{1, 2, \ldots, |V|\}$.

La **matrice di adiacenza** di G è la matrice simmetrica `M` con $|V| \times |V|$ valori booleani in cui `M[x][y] == 1` se e solo se l'arco $(v_{\mathtt{x}}, v_{\mathtt{y}}) \in E$.

La rappresentazione con matrici di adiacenza sono accettabili solamente se i grafi da manipolare *non* sono *sparsi* (A G mancano "pochi" archi per essere il grafo completo su V).

Lo spazio necessario con questa rappresentazione è $O(V^2)$. Anche il tempo richiesto da molti degli algoritmi fondamentali è $O(V^2)$ con questa rappresentazione.

Liste d'adiacenza.

Quando il grafo è sparso, diciamo $|E| = O(|V| \log |V|)$, la rappresentazione attraverso **liste di adiacenza** è spesso da preferire.

In questa rappresentazione vi è una lista per ogni vertice $v_i \in V$.
Ogni elemento di una tale lista è a sua volta un vertice $v_j \in V$.
v_j appartiene alla lista associata a v_i se e solo se $(v_i, v_j) \in E$.

I dettagli implementativi variano a seconda delle necessità applicative.
Ad esempio, le liste di adiacenza possono essere contenute in un array statico o dinamico, una lista concatenata, una tabella hash, un albero di ricerca, etc...
Anche le informazioni contenute nei nodi delle liste di adiacenza dipendono dalle esigenze dell'applicazione.

Consideriamo una semplice implementazione con un array allocato in memoria dinamica di liste d'adiacenza.

Questa implementazione non è adatta ad applicazioni che modifichino frequentemente i grafi.

```
struct node {              /* nodo di lista di adiacenza */
    int v;
    struct node *next;
};

struct graph {             /* struttura associata a ogni grafo */
    int V;                 /* numero nodi  */
    int E;                 /* numero archi */
    struct node **A;       /* array di liste di adiacenza */
};
```

Creazione del grafo

```
struct graph *creategraph(int nv, int ne)
{
    struct graph *g = malloc(sizeof(struct graph));

    if(!g) {
        fprintf(stderr,"Errore di Allocazione\n");
        exit(-1);
    }
    g->E = ne;
    g->V = nv;
    if(!(g->A = calloc(nv,sizeof(struct node *))))  {
        fprintf(stderr,"Errore di Allocazione\n");
        exit(-2);
    }
    return g;
}
```

La funzione `creategraph` richiede il numero di vertici e il numero di nodi del grafo da creare. Alloca spazio per ogni lista di adiacenza, inizialmente vuota.

Lettura del grafo:

Dopo aver creato la struttura delle liste di adiacenza, bisogna inserire gli archi nelle liste stesse.

```
void readgraph(struct graph *g, FILE *fp) {
    int i,v1, v2;

    for(i = 0; i < g->E; i++) {
        fscanf(fp,"%d %d",&v1,&v2);
        g->A[v1-1] = vertexinsert(g->A[v1-1],v2);
        g->A[v2-1] = vertexinsert(g->A[v2-1],v1);
    }
}
```

Usiamo questa funzione che legge da un file gli archi. Nel file ogni riga contiene un unico arco specificato come coppia di vertici.

La funzione per inserire i vertici nelle liste di adiacenza è una normale funzione per l'inserzione in testa a liste concatenate:

```
struct node *vertexinsert(struct node *p, int k) {
    struct node *q = malloc(sizeof(struct node));

    if(!q) { fprintf(stderr,"Errore di Allocazione\n"); exit(-3); }
    q->v = k;
    q->next = p;
    return q;
}
```

Si noti che per ogni arco si devono eseguire due chiamate a `vertexinsert` su due liste diverse dell'array `g->A`.
Dato un arco $(v, w) \in E$, si deve inserire v nella lista associata a w, così come w nella lista associata a v.

Attraversamento in Profondità

L'attraversamento in profondità (**depth-first search, DFS**) di un grafo non orientato consiste nel visitare ogni nodo del grafo secondo un ordine compatibile con quanto qui di seguito specificato:

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.

Implementiamo la visita DFS tramite una semplice funzione ricorsiva:

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

void dfs(struct graph *g) {
    int i, *aux = calloc(g->V,sizeof(int));
    if(!aux) { fprintf(stderr,"Errore di Allocazione\n"); exit(-4); }
    for(i = 0; i < g->V; i++)
        if(!aux[i]) {
            printf("\n%d,",i+1);
            dfs1(g,i,aux);
        }
    free(aux);
}
```

La procedura ricorsiva implementata per la visita DFS usa un array di appoggio per memorizzare quando un vertice è già stato incontrato.

Quando `dfs1` è richiamata da `dfs` si entra in una nuova componente connessa.

`dfs1` richiamerà se stessa ricorsivamente fino a quando tutta la componente è stata visitata.

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

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.

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

void bfs(struct graph *g) {
    int i, *aux = calloc(g->V,sizeof(int));
    if(!aux) { fprintf(stderr,"Errore di Allocazione\n"); exit(-4); }
    for(i = 0; i < g->V; i++)
        if(!aux[i]) {
            printf("\n%d,",i+1);
            bfs1(g,i,aux);
        }
    free(aux);
}
```

La procedura implementata per la visita BFS usa un array di appoggio 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 $O(|V| + |E|)$.