ALGORITMI E STRUTTURE DATI

Esercitazioni

Andrea Orlandini
http://www.dia.uniroma3.it/~orlandin/asd/
e-mail: orlandin@dia.uniroma3.it

Orario di ricevimento: Martedì 14.00 - 16.00

Puntatori e Grafi

StudentiDIA Forum

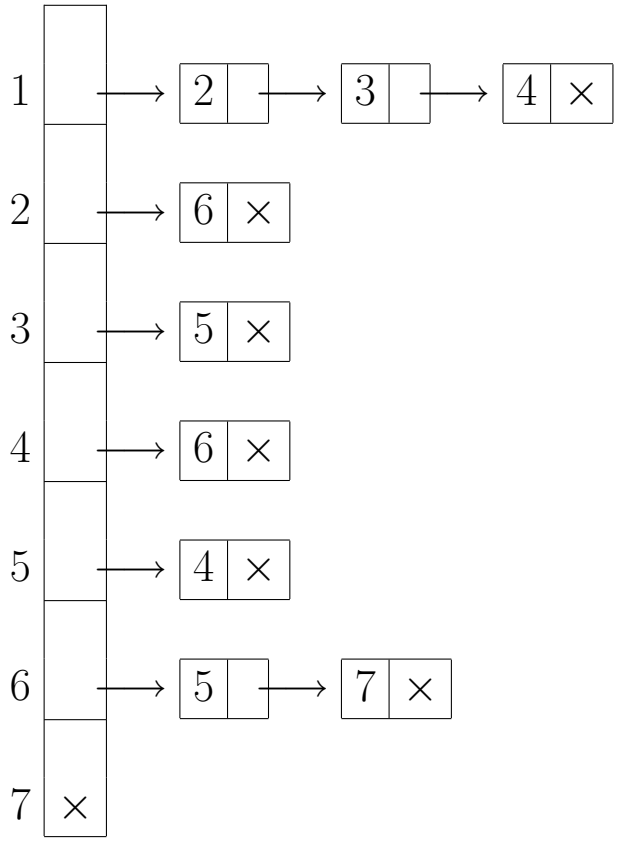
http://forum.studentidia.org/

Dove trovare i lucidi presentati a lezione

http://limongelli.dia.uniroma3.it/asd/

Implementazione dei Grafi

- Rappresentazione mediante vettore delle liste di successori: un vettore di n elementi ed n liste



```
#define NumNodi ...

typedef int TipoNodo;

struct RecordListaSucc {
  TipoNodo successore;
  struct RecordListaSucc *next;
};
typedef struct RecordListaSucc *TipoListaSucc;

struct Grafo {
  TipoListaSucc vett_lista_succ[NumNodi];
};

typedef struct Grafo *TipoGrafo;
```

- TipoGrafo è un puntatore ad un vettore di tipo TipoListaSucc
- i-esimo elemento del vettore è il puntatore alla lista dei successori del nodo *i*

Implementazione delle funzioni fondamentali

- Un grafo vuoto è rappresentato da un vettore i cui elementi sono tutti puntatori nulli
- Una funzione per inizializzare la struttura può quindi essere la seguente:

```
TipoGrafo InitGrafo()
{
  TipoNodo i;
  TipoGrafo grafo;

  grafo = malloc(sizeof(struct Grafo));

  for (i = 0; i < NumNodi; i++)
    grafo->vett_lista_succ[i] = NULL;

  return grafo;
}
```

- Le operazioni per verificare l'esistenza e effettuare l'inserimento o la cancellazione di un arco si implementano tramite le relative operazioni sulle liste
- Verificare se un arco $<i,j>$ è presente in un grafo equivale a verificare la presenza di j nei successori di i

Test di Esistenza di un Arco

- Definisco una funzione che effettua il test di presenza di un elemento in una lista e la richiamo nella funzione TestEsisteArco

```
bool TestElementoInLista(TipoListaSucc lista, TipoNodo j)
{
  if (lista == NULL)
    return FALSE;
  else
    if (lista->successore == j)
      return TRUE;
    else
      return (TestElementoInLista(lista->next, j));
}


bool TestEsisteArco(TipoGrafo grafo, TipoNodo i, TipoNodo j)
{
  return (TestElementoInLista
                  (grafo->vett_lista_succ[i], j));
}
```

Inserimento di un arco

- l'inserimento di un arco è effettuato aggiungendo un elemento nella lista dei successori
  - Controllo se l'arco è già presente nel grafo
  - Nel caso in cui l'arco deve essere effettivamente inserito devo allocare lo spazio per il nuovo elemento nella lista dei successori
- Il successore è inserito in testa

```
TipoGrafo InserArco(TipoGrafo grafo, TipoNodo i, TipoNodo j)
{
  TipoListaSucc lista;

  if (!TestEsisteArco(grafo, i, j)) {
    lista = malloc(sizeof(struct RecordListaSucc));
    lista->successore = j;
    lista->next = grafo->vett_lista_succ[i];
    grafo->vett_lista_succ[i] = lista;
  }
  return grafo;
}
```

Rimozione di un arco

- Analogamente, rimuovere un arco consiste nel cancellare un elemento dalla lista dei successori
  - Libero la zona di memoria occupata dall'elemento nella lista

```
TipoListaSucc EliminaDaLista(TipoListaSucc lista, TipoNodo j) {
  TipoListaSucc lista_aux;

  if (lista != NULL) {
    if (lista->successore == j) {
      lista_aux = lista;
      lista = lista->next;
      free(lista_aux);
    }
    else
      lista->next=EliminaDaLista(lista->next, j);
  }
  return lista;
}

TipoGrafo ElimArco(TipoGrafo grafo, TipoNodo i, TipoNodo j)
{
  grafo->vett_lista_succ[i] = EliminaDaLista(grafo->vett_lista_succ[i], j);
  return grafo;
}
```

Algoritmi di visita dei Grafi

- Visita in ampiezza (Breadth First Search): visito un nodo e poi i suoi successori

```
void VisitaInAmpiezza(TipoGrafo grafo, TipoNodo nodo_iniz)
{
  TipoNodo i, j;
  TipoCoda nodi_da_visitare;
  bool nodi_visti[NumNodi];

  for (i = 0; i < NumNodi; i++) nodi_visti[i] = FALSE;

  InitCoda(&nodi_da_visitare);
  InCoda(&nodi_da_visitare, nodo_iniz);

  while (!TestCodaVuota(nodi_da_visitare)) {
    OutCoda(&nodi_da_visitare, &i);
    if (!nodi_visti[i]) {
      Analizza(i); nodi_visti[i] = TRUE;
>     for (j = 0; j < NumNodi; j++)                                <
>       if (TestEsisteArco(grafo, i, j) && !nodi_visti[j])         <
>         InCoda(&nodi_da_visitare, j);                            <
     }
   }
}
```

Inserimento in coda dei successori più efficiente

- L'inserimento dei successori in coda avviene tramite un ciclo non efficiente
  - Verifica per ogni nodo se esiste un arco proveniente dal nodo attuale che non è stato ancor visitato
- Sfruttiamo la rappresentazione per liste di successori
- papp è di tipo TipoListaSucc

```
>     papp=grafo->vett_lista_succ[i];                    <
>     while (papp!=NULL) {                               <
>       if (!nodi_visti[papp->successore])               <
>         InCoda(&nodi_da_visitare,papp->successore);<
>       papp=papp->next;}                                <
```

- Verifichiamo se tra i successori del nodo attuale qualcuno è da visitare
- La visita viene fatta rispettando l'ordinamento della lista dei successori invece di provare in maniera esaustiva tutti i nodi del grafo.

Visita In Profondità (1)

- Visita in profondità Ricorsiva (Depth First Search)
- Visito subito i successori che incontro
- Inizializzo tutte le etichette di visita a FALSE
- Richiamo VisitaRicorsiva sul grafo a partire dal nodo iniziale

```
bool nodi_visti[NumNodi];

void VisitaRicInProfondita(TipoGrafo grafo, TipoNodo nodo_iniz)
{
  TipoNodo i;

  for (i = 0; i < NumNodi; i++)
    nodi_visti[i] = FALSE;
  VisitaRicorsiva(grafo, nodo_iniz);
}
```

Visita In Profondità (2)

```
void VisitaRicorsiva(TipoGrafo grafo, TipoNodo i)
{
  TipoNodo j;

  if (!nodi_visti[i]) {
    Analizza(i);nodi_visti[i] = TRUE;

    for (j = 0; j < NumNodi; j++)
      if (TestEsisteArco(grafo, i, j)&&!nodi_visti[j])
        VisitaRicorsiva(grafo, j);
  }
}
```

- anche qui possiamo fare la stessa considerazione sull'efficienza di prima

```
>     papp=grafo->vett_lista_succ[i];                <
>     while (papp!=NULL) {                           <
>       if (!nodi_visti[papp->successore])           <
>         VisitaRicorsiva(grafo, papp->successore);  <
>       papp=papp->next;}                            <
```

Verifica di Esistenza Cammino

- Come al solito per la risoluzione di problemi si utilizzano algoritmi basati su algoritmi di visita
- Implementiamo una funzione che verifica l'esistenza di un cammino da un nodo ad un'altro

```
Input: G:Grafo,i:Nodo,j:Nodo

Output: TRUE se j è raggiungibile da i,
        FALSE altrimenti
```

- Molto semplicemente possiamo implementare la funzione Analizza presente nella funzione di visita con queste caratteristiche:
  - Analizza avrà come parametri di INPUT le etichette del nodo visitato e del nodo j da raggiungere
  - Nel caso coincidano significa che esiste il cammino che cerchiamo
  - Se al termine della ricerca non troviamo il nodo da raggiungere significa che non è esiste un cammino dal nodo i al nodo j

```
bool Analizza(TipoNodo visitato, TipoNodo daTrovare) {
  if (visitato = daTrovare)
    return TRUE;
  else
    return FALSE;
}
```

  - Per esercizio integrare in un algoritmo di visita questa funzione Analizza

Esercizio

Dato un grafo G, un nodo X non presente in G e una serie di archi tra X e i nodi di G e/o viceversa, scrivere un algoritmo per costruire un nuovo grafo G' contenente il nuovo nodo e i nuovi archi.

Per semplicità:

- Etichette -> interi (N elementi in G e N+1 elementi in G')
- I nuovi archi sono già indicati tenendo conto della indicizzazione di cui sopra

Esercizio

Dato un grafo G, un nodo X non presente in G e una serie di archi tra X e i nodi di G e/o viceversa, scrivere un algoritmo per costruire un nuovo grafo G' contenente il nuovo nodo e i nuovi archi.

Come al solito ragioniamo TOP-DOWN

1. Creo un nuovo grafo G' copiando i nodi di G e aggiungendo un nuovo nodo
2. Copio tutti gli archi di G in G'
3. Aggiungo in G' i nuovi archi

```
PROBLEMA(G,LISTAARCHI)
  G' = COPIANODI(G)
  COPIAARCHI(G,G')
  AGGIUNGIARCHI(G',LISTAARCHI)
  RETURN G'
```

COPIANODI

- Siano N i nodi di G, assegno al nuovo nodo in G' la posizione N+1
- Quindi creo un nuovo vettore con N+1 elementi per G'

```
COPIANODI(G,G')
  N <- LENGTH(G)
  G'[N+1]
  RETURN G'
```

COPIAARCHI

- Copio le liste di successori di ogni elemento di G in G'

```
COPIAARCHI(G,G')
  G'[N+1] <- NIL
  FOR I = 1 TO N
    G'[I] <- COPIALISTA(G[I])

COPIALISTA(L)
  ;; ALLOCO MEMORIA PER LISTASUCC
  P <- LISTASUCC
  WHILE (L <> NIL)
    ;; ALLOCO MEM PUNTATA DA NEXT[P]
    P <- NEXT[P]
    SUCCESSORE[P] <- SUCCESSORE[L]
    L <- NEXT[L]
  NEXT[P] <- NIL
  P <- LISTASUCC
  LISTASUCC <- NEXT[LISTASUCC]
  ;; LIBERO MEMORIA P
  RETURN LISTASUCC
```

- Così ho terminato la copia del grafo G con l'aggiunta dell'elemento N+1
- Ora devo inserire gli archi in G'

AGGIUNGIARCHI

- Ipotizzo che i nuovi archi da inserire siano rappresentati mediante una lista di coppie:
  - A -> B è rappresentato da un elemento della lista che contiene un campo SORGENTE con valore pari a A e un campo DESTINAZIONE con valore pari a B
- Per ogni coppia che incontriamo dobbiamo semplicemente inserire un arco nel grafo (algoritmo già visto prima)

```
AGGIUNGIARCHI(G,LA)
  WHILE (LA <> NIL)
    INSCERISCIARCO(G,SORGENTE[LA],DESTINAZIONE[LA])
    LA <- NEXT[LA]
```

- Essendo sicuri che non sia presente l'arco da inserire possiamo riscrivere l'algoritmo evitando il controllo di presenza dell'arco