Algoritmi con alberi

(Tree-Search e strutture dati)

Alberto Cicciarelli, seminario teoria dei grafi 2020

Tree-Search: in cosa consistono?

Gli algoritmi Tree-Search sfruttano le proprietà degli alberi costruiti all'interno di un grafo G (generanti o non) attraverso le quali risulta molto più veloce studiare le caratteristiche del grafo in esame.
Stabilire per esempio se un grafo sia connesso o meno può essere un facile esercizio per grafi di piccola taglia ma risulta estremamente complicato e dispendioso non appena il numero di nodi comincia a crescere.

Costruire un albero in G

Sia T un albero contenuto in G:

- Se V(T)=V(G) siamo in presenza di un albero generante e valgono tutte le proprietà di questi
- Se |V(T)|<|V(G)| allora il taglio ∂(T) può essere nullo (grafo non connesso) o meno. Se ∂(T) non è nullo allora per ogni $\underline{xy}$ Є ∂(T) con x in V(T) ed y in V(G)-V(T) possiamo aggiungere il lato $\underline{xy}$ ed il vertice y a T ottenendo ancora un albero.
  Se T di partenza è un vertice isolato possiamo costruirci un albero a partire da quest'ultimo.

La ricerca di componenti connesse

Due algoritmi in particolare sfruttano la costruzione di alberi all'interno di un grafo per determinare se questo sia connesso o meno, restituendo anche ulteriori informazioni:

- *Breadth-First-Search*, è un algoritmo con cui è possibile studiare le componenti connesse di un grafo e la distanza tra i vertici
- *Depth-First-Search*, è un algoritmo con cui è possibile studiare le componenti connesse di un grafo ed i suoi vertici di taglio

Grafi semplici e non

Per questo seminario la nostra attenzione sarà posta sui grafi semplici, tale scelta permette una trattazione più fluida ed intuitiva dell'argomento. Inoltre i grafi semplici riescono con facilità a rappresentare network reali senza dover ricorrere ad ulteriori elementi (loops, multilinks, ecc.) che appesantirebbero lo studio di situazioni pratiche.
Comunque l'estensione ad altre tipologie di grafi non risulta particolarmente complicata e tiene conto delle stesse idee.

Breadth-First Search

L'algoritmo Breadth-First Search (BFS) è un algoritmo in grado di fornire, a partire da un grafo G e un nodo di partenza *i*, un albero T di G, una lista delle distanze tra i e tutti i nodi raggiungibili da questo.
Qui proponiamo una versione del BFS che permette anche di ricostruire il cammino più corto fra un nodo e l'altro.

Breadth-First Search

L'algoritmo lavora nel seguente modo:

- Viene scelto un vertice *i* di partenza (radice)
- Ogni vicino di *i* viene aggiunto ad una lista (mark) e posto a distanza 1 costruendo una funzione (*l*: V—>N) che associa ad ogni vertice la distanza dalla radice, con predecessore *i*
- Per ogni nodo *v* aggiunto in mark vengono aggiunti a tale lista i vicini di *v* che non sono già presenti e posti a distanza 2 con predecessore *v*
- Si itera il procedimento per tutti i nodi raggiungibili in questo modo

Breadth-First Search

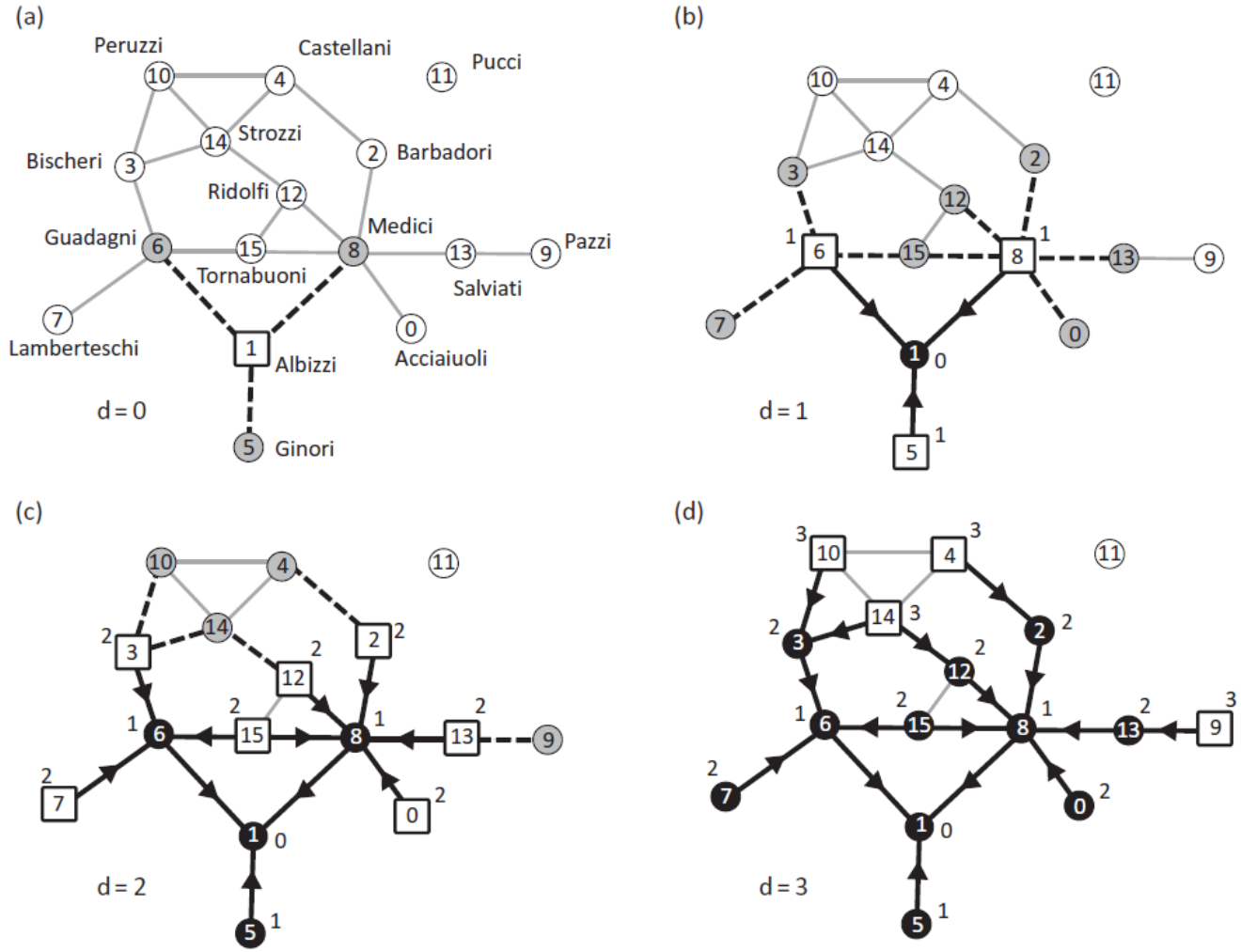
Algorithm 14 `BFS()`

Input: G (unweighted graph), i
Output: distances from i to all the other nodes in G, and list of predecessors

```
1:  for j = 0 to N-1 do
2:      dist[j] ← N
3:      marked[j] ← N+1
4:  end for
5:  dist[i] ← 0
6:  marked[0]← i
7:  d ← 0
8:  n ← 0
9:  nd ← 1
10: ndp ← 0
11: while d<N and nd >0 do
12:     for k=n to n + nd - 1 do
13:         cur_node ← marked[k]
14:         for all j in neigh[cur_node] do
15:             if dist[j] = d+1 then
16:                 add_predecessor(j,k,preds)
17:             end if
18:             if dist[j] = N then
19:                 dist[j] ← d+1
20:                 add_predecessor(j,k,preds)
21:                 marked[n + nd + ndp] ← j
22:                 ndp ← ndp + 1
23:             end if
24:         end for
25:     end for
26:     n ← n + nd
27:     nd ← ndp
28:     ndp ← 0
29:     d ← d+1
30: end while
31: return dist, preds
```

Breadth-First Search

In figura: Esempio, applicazione del BFS ad un network reale (famiglie fiorentine collegate da matrimoni nel rinascimento)



Breadth-First Search

Proprietà

Teorema 6.2 B-M. Sia T un albero ottenuto da un grafo connesso G a partire dalla radice r tramite il BFS, allora:

- a) Per ogni vertice v di G, $l(v)=d_T(r,v)$ con $d_T(r,v)$ distanza tra radice r e vertice sull'albero (livello)
- b) Ogni lato di G comprende vertici sugli stessi livelli, o consecutivi di T, quindi $|l(u)-l(v)|\leq 1$, per ogni $\underline{uv}$ in E(G)

Teorema 6.3 B-M. Sia G un grafo connesso. Allor i valori della funzione di livello l che sono ottenuti dal BFS sono proprio le distanze, in G, dei vari vertici dalla radice r:

$$l(v)= d_T(r,v)= d_G(r,v) \text{ per tutti i } v \text{ in } V(G)$$

Oss. È possibile modificare l'algoritmo in modo da ottenere il numero di componenti connesse

Breadth-First Search

Efficienza

È possibile provare che l'algoritmo BFS così descritto è ottimale, ovvero non è possibile trovare un algoritmo più efficiente in grado di visitare tutti i nodi, e tener traccia delle distanze, a partire da un singolo nodo.
Il numero di iterazioni che comporta è circa O(N+K), perché l'algoritmo ha bisogno di scorrere tutti i nodi ed i lati del grafo esattamente una volta. Se poi lo vogliamo ripetere per tutti i nodi avremo O(N(N+K)).

Depth-First Search

Il Depth-First Search (DFS) è un algoritmo in grado di stabilire se un grafo è connesso o meno. La strategia consiste, dato un punto di partenza, nell'esplorare «in profondità» un albero T incluso in un grafo G attraverso i primi vicini.
Qui presentiamo una sua versione in forma ricorsiva che ci permette, in poche righe, di calcolare il numero di componenti connesse di G.

Depth-First Search

I passaggi chiave dell'algoritmo sono i seguenti:

- Scegliere un nodo *i* appartenente a G
- Aggiungere il primo vicino di *i* ad un albero T (formato da *i*)
- Per ogni nodo adiacente ad i procedere aggiungendo all'albero il suo primo vicino, controllando che non sia già inserito, così via per ogni nodo aggiunto all'albero
- Quando arriviamo al punto in cui un nodo non ha nessun vicino proseguire a ritroso esplorando i restanti nell'albero T

Depth-First Search

Algorithm 18 `DFS()`

Input: ic, i, nc, f

Output: s{size of the tree starting at i}

```
1:  ic[i] ← nc
2:  s ← 1
3:  for all j in neigh[i] do
4:      if ic[j] = 0 then
5:          s ← s + DFS(ic,j,nc,f)
6:      end if
7:  end for
8:  f[time] ← i
9:  time ← time + 1
10: return s
```

Depth-First Search

La funzione così scritta presenta un passaggio opzionale, la lista f[] e la variabile time. Tali righe se aggiunte al codice permettono di tener conto l'ordine con cui viene composto l'albero e risalire al tempo in cui viene inserito un nodo all'interno di questo.
La lista f[] così presentata può essere ricondotta alla funzione *f()* presente nel B-M.

Depth-First Search

Ricerca di componenti

Notiamo che l'algoritmo scritto nelle slide precedenti non ci dà informazioni su tutte le eventuali componenti connesse, ma ci restituisce informazioni esclusivamente su un albero T incluso nel grafo di partenza. Per avere una maggiore descrizione di tutte le componenti connesse è necessario inserire l'algoritmo come funzione all'interno di un altro listato.

Depth-First Search

Ricerca di componenti

Algorithm 19 `components()`

Input: G

Output: ic,nc,sizes, f

```
1:  for i = 0 to N-1 do
2:      ic[i] ← 0
3:      f[i] ← 0          ← Opzionale
4:  end for
5:  nc ← 0
6:  time ← 0              ← Opzionale
7:  for i = 0 to N-1 do
8:      if ic[i] = 0 then
9:          nc ← nc + 1
10:         s ← DFS(ic,i,nc,f)
11:         sizes[nc] ← s
12:     end if
13: end for
```

Depth-First Search

Efficienza

Osserviamo che l'algoritmo DFS così implementato visita ogni nodo ed ogni link esattamente una volta quindi il nostro tempo di svolgimento in iterazioni sarà O(N+K), ovvero lo stesso del BFS. Tuttavia l'algoritmo DFS risulta più compatto, soprattutto se scritto in una forma ricorsiva.

Depth-First Search

Vertici di taglio

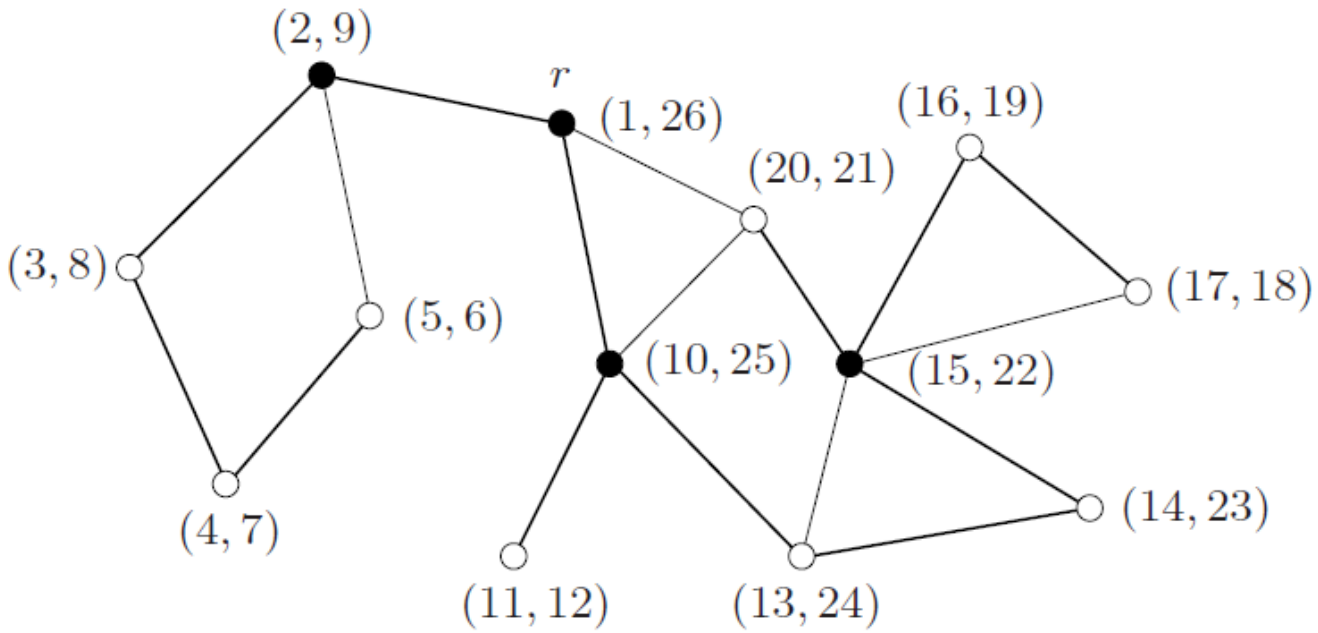
Tramite il teorema che segue è possibile trovare vertici di taglio utilizzando l'algoritmo DFS, apportando modifiche a quanto esposto in precedenza.

Teorema 6.7 B-M: Sia T un albero generato dall'algoritmo DFS per un grafo connesso. La radice *r* di T è un vertice di taglio se e solo se ha almeno due «figli». Ogni altro vertice *v* di T è di taglio se e solo se ammette almeno un figlio tale che ogni altro vertice «discendente» da questo non è collegato a nessun «antenato» di *v* tramite lati del coalbero.

Depth-First Search

Vertici di taglio

In figura: r è la radice dell'albero, i lati evidenziati sono quelli dell'albero DFS e i vertici colorati di nero sono quelli di taglio.



Commenti e listati in C

Avendo illustrato il generale comportamento degli algoritmi BFS e DFS, assieme alle loro proprietà, vediamo come questi possono essere scritti in C, commentando alcuni risultati di simulazioni effettuate

Commenti e lisati in c

BFS

```c
//DEFINIZIONE DELLA FUNZIONE **BFS**
void BFS(int *A, int *dist, int **pre, int n, int P) {
  int ii, ndp=0, i, d=0, nd=1, m=0, *mark, k=0;
  mark= (int *)calloc(1,sizeof(int)); //inizializzo
  mark[0]=P;
  for(i=0; i<n; i++){
    dist[i]=n;
    *(pre+i)=(int*) calloc(1,sizeof(int));
    }
    dist[P]=0;

  while(nd!=0){
    for(i=m; i<(m+nd); i++){
      for(ii=0;ii<n;ii++){
        if(A[n*mark[i]+ii]==1) {
          if(dist[ii]==(d+1)) {
            k= pre[ii][0]; //alloco pre1
            k++;
            *(pre+ii)= (int*) realloc(*(pre+ii), (k+1)*sizeof(int));
            pre[ii][0]=k;
            pre[ii][k]=mark[i];
            k=0;
            }
```

Commenti e lisati in c

BFS

```
          if(dist[ii]==n){
            dist[ii]=d+1; //alloco dist
            ndp++;
            mark= (int*) realloc(mark, (m+nd+ndp)*sizeof(int)); //alloco mark
            mark[m+nd+ndp-1]=ii;
            k= pre[ii][0]; //alloco pre1
            k++;
            *(pre+ii)= (int*) realloc(*(pre+ii), (k+1)*sizeof(int));
            pre[ii][0]=k;
            pre[ii][k]=mark[i];
            k=0;
          }
        }
      }
    }
    m+=nd;
    nd=ndp;
    ndp=0;
    d++;
  }

free(mark);
}
```

Commenti e lisati in c

DFS

```c
//DEFINIZIONE **DFS**
int DFS(int *ic, int i, int **J, int nc){
  int s=1, k, j, ii;
  ic[i]=nc;
  k=J[i][0];
  for(j=1; j<=k; j++){
    ii=J[i][j];
    if(ic[ii]==0){
      s+=DFS(ic, ii, J, nc);
    }
  }
  return s;
}

//DEFINIZIONE **componenti**
int componenti(int **J, int n){
  int nc, *ic, *size, i, jj, s;
  ic= (int*) calloc(n,sizeof(int));
  size= (int*) calloc(1,sizeof(int));
  for(jj=0;jj<n;jj++){
    ic[jj]=0;
  }
  nc=0;
  for(i=0;i<n;i++){
    if(ic[i]==0){
      nc++;
      s= DFS(ic, i, J, nc);
      if(nc>1){
        size= (int*) realloc(size, nc*sizeof(int));
        size[nc-1]=s;
      }else{size[0]=s;}
    }
  }
  return nc;
}
```

Commenti e listati in c

Speedtest

Modificando il primo listato, riferito al BFS, è possibile ottenere una versione light dell'algoritmo (trascurando la lista dei predecessori) da inserire come funzione all'interno di un'altra che conti le componenti connesse.
Così facendo è possibile realizzare un programma che confronti la velocità di esecuzione dei due algoritmi nel contare le componenti connesse di un network reale. Secondo quanto scritto in precedenza dovremmo avere tempi di esecuzione simili.

Commenti e listati in c

Speedtest

```
alberto@LAPTOP-M0DPCUM3:~$ gcc SpeedTest.c -o Stest.exe
alberto@LAPTOP-M0DPCUM3:~$ time ./Stest.exe coauthorship/coauthorship_astro-ph/astro-ph_names.txt coauthorship/coauthorship_astro-ph/astro-ph.net
NUMERO DI NODI= 16706
NUMERO DI LINKS= 121251

OK J
Numero di componenti connesse= 1029
real    0m3.006s
user    0m1.266s
sys     0m1.734s
alberto@LAPTOP-M0DPCUM3:~$ gcc SpeedTest.c -o Stest.exe
alberto@LAPTOP-M0DPCUM3:~$ time ./Stest.exe coauthorship/coauthorship_astro-ph/astro-ph_names.txt coauthorship/coauthorship_astro-ph/astro-ph.net
NUMERO DI NODI= 16706
NUMERO DI LINKS= 121251

OK J
Numero di componenti connesse= 1029
real    0m3.096s
user    0m1.250s
sys     0m1.813s
alberto@LAPTOP-M0DPCUM3:~$
```

Usando BFS

Usando DFS

Oltre che per lo studio delle reti reali è possibile costruire algoritmi che sfruttino le proprietà degli alberi per gestire strutture dati in maniera efficiente. Ne è un esempio l'algoritmo Binary-Search Tree (BST) che verrà qui esposto.

Alberi per strutture dati

Alberi per strutture dati *BST*

Il BST è un algoritmo che permette di costruire un albero binario contenenti dati presenti in una certa struttura, ad esempio un array. Il vantaggio di usare questa rappresentazione è quello di poter gestire in maniera ottimale operazioni di ricerca per un certo elemento all'interno di una struttura dati.
Si pensi al check che bisogna effettuare per inserire un elemento in una lista senza ripeterlo.

Alberi per strutture dati
BST

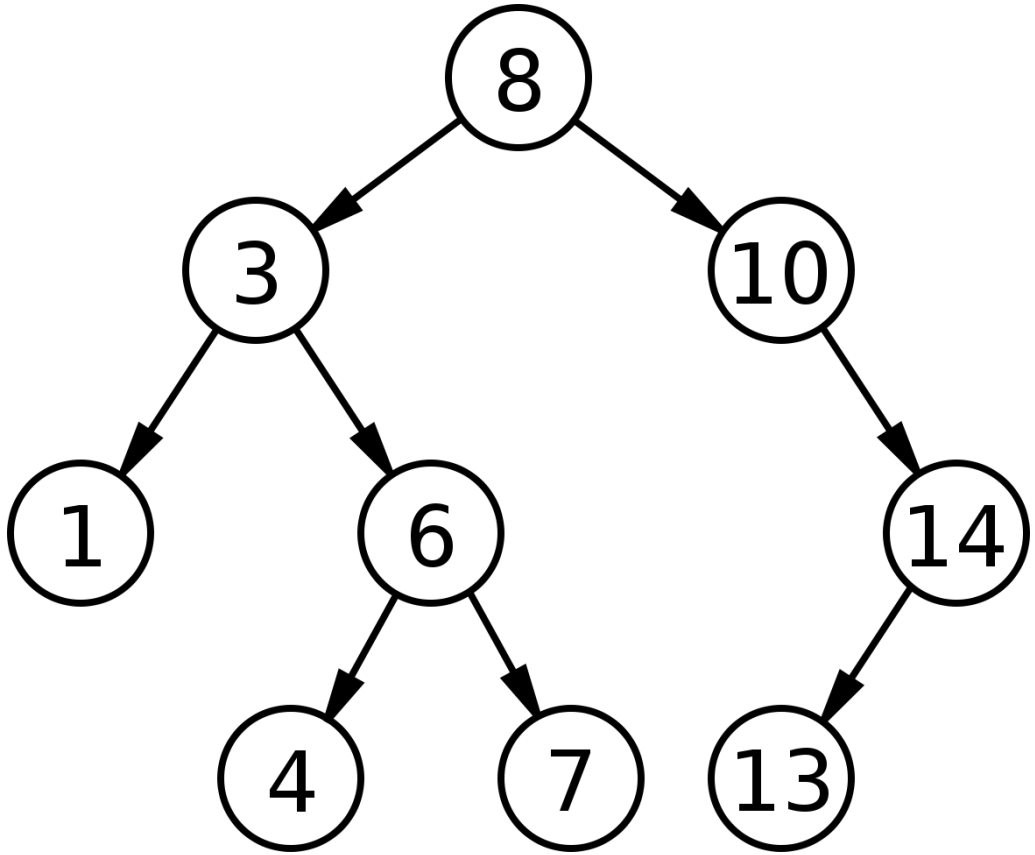
I passaggi chiave di tale algoritmo sono:

- Viene data una certa struttura dati, per comodità si consideri composta da numeri
- Ad ogni elemento x della struttura viene associato uno ed un solo nodo dell'albero attraverso un numero, *key*=x
- Scegliere un unico elemento di partenza, *root*
- Ogni nodo, esclusa la *root*, ha un solo *genitore* e due *discendenti*
- Il *discendente* di destra è un numero maggiore del *genitore*, quello di sinistra è un numero minore

Alberi per strutture dati

BST

In figura: un esempio grafico di un albero costruito tramite l'algoritmo BST partendo da una certa struttura dati.



Alberi per strutture dati
BST, proprietà

- Una volta costruito l'albero e riempito dei nostri dati è possibile scrivere funzioni che in maniera ottimale operino sulla nostra struttura.
- Si pensi al seguente problema: *aggiungere un elemento x ad un array unidimensionale lungo N senza ripeterlo*.
  Normalmente dovremmo scorrere l'array, verificare che l'elemento x non sia presente e poi inserirlo. Un operazione che ci costa N passi. Se invece rappresentiamo il nostro array sotto forma di albero e scriviamo una funzione che ci cerca x in maniera ottimale, basandosi sulle disuguaglianze che hanno generato l'albero, possiamo effettuare il check con un tempo di esecuzione O(logN), riducendo di molto il costo computazionale dell'operazione.

Alberi per strutture dati

BST, implementazione

- Per implementare l'algoritmo BST può essere scelto il linguaggio C usando *struct* e funzioni ricorsive.
- Nell'esempio proposto sono state scritte 3 funzioni, due riguardanti la generazione dell'albero a partire da certi elementi ed una riguardante la ricerca su tale albero di un elemento. Per costruire l'albero si è fatto uso di una funzione *insert* che verifichi dove inserire il dato e di una funzione *newNode* che aggiunga effettivamente tale dato all'albero. La funzione *search* invece ci restituisce un certo valore a seconda che un certo elemento sia uguale o meno ad uno presente nell'albero.

Alberi per strutture dati

BST, implementazione

```c
//DEFINISCO FUNZIONI PER **BST** (struct)
struct node {
    int key;
    struct node *left, *right;
};

struct node *newNode(int item)
{
    struct node *temp =  (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

struct node* insert(struct node* node, int key, int *search) {
    if(node==NULL) {
      return newNode(key);
    }
    if(key==node->key){
      search[0]=1;
    }
    if(key < node->key){
          node->left = insert(node->left, key, search);
        }
    if (key > node->key){
          node->right = insert(node->right, key, search);
        }
    return node;

}
```

Alberi per strutture dati

BST, implementazione

```
int search(struct node* node, int key) {
    int A;
    if(node==NULL) {
      return 0;
    }
    if(key==node->key){
      return 1;
    }
    if(key < node->key){
          A=search(node->left, key);
        }
    if (key > node->key){
          A=search(node->left, key);
        }
    return A;

}
```

Fonti utilizzate

Per questa presentazione sono stati utilizzati come fonti i seguenti libri:

- Graph Theory, Bondy-Murty, i.e. capitolo 6
- Complex Networks, Latora-Nicosia-Russo, i.e. appendici 3, 6, 8