
**di Arcadio Ciro & Paolo Cittadini (gruppo 38)**

# ArCiSoft's LinkEarth v1.0

**elaborato per il corso di “Laboratorio di Algoritmi e Strutture Dati”, presso l'università Federico II, c.d.l in Informatica.**
**Proff. F.Cutugno e L.Lamberti.**

## Documentazione Esterna, relazione e Manuale d'uso.

# 0.0.Indice

# 1.Introduzione

### *1.0.Background del problema*

Si vuole procedere nell'implementazione di un programma software che operi su varie strutture dati derivate, quali alberi binari e grafi.
Il nostro scopo primario è quello di realizzare, partendo da una fonte dati, un albero binario perfettamente bilanciato, con un minimo di 5k nodi.
La fonte dati è da considerarsi, secondo le direttive del committente, di libera scelta.
Costruito l'albero binario è necessario fornire il set di function primario per la gestione della suddetta struttura dati, ad esempio l'implementazione di una visita in ampiezza.
É richiesta inoltre l'implementazione di un algoritmo in grado di estrarre in ordine i nodi di un dato livello (è fatta esplicita richiesta del livello (n-1)esimo).
A partire dai nodi così estratti è doveroso procedere nella creazione di un grafo che una volta implementato, sarà alla base nel nostro progetto.
La struttura dati Grafo, è alla base della teoria delle Social Network, in questo caso Small Word Network, che a loro volta sono i mattoni fondamentali delle teorie sulla comunicazione globale (navale, aeronautica, ferroviaria, etc.).
Con il nostro Grafo popolato dobbiamo costruire un sistema di gestione e di esplorazione di Small Word Network:
Inserire collegamenti di tipo “LOCALS”, fra nodi vicini, e link “Long Distance” che colleghino nodi siti a distanze elevate.
É richiesto lo sviluppo di funzioni per la gestione di nodi di tipo HUB, cioè nodi aventi una quantità tale di link Long Distance, da renderli nuclei di comunicazione di rilevante importanza.
Infine i dati da noi raccolti, in merito alla tempistica di esplorazione del grafo, in base al tipo di gestione della rete, devono essere utilizzati per il calcolo della complessità sperimentale e computazionale.

Detto questo tocca a noi, i “programmatori”, trovare l'applicazione dell'elenco di richieste (molto esplicite) di cui sopra, ad un problema che rassomigli in positivo ad una reale applicazione della teoria S.W.N.
Segue la nostra idea.

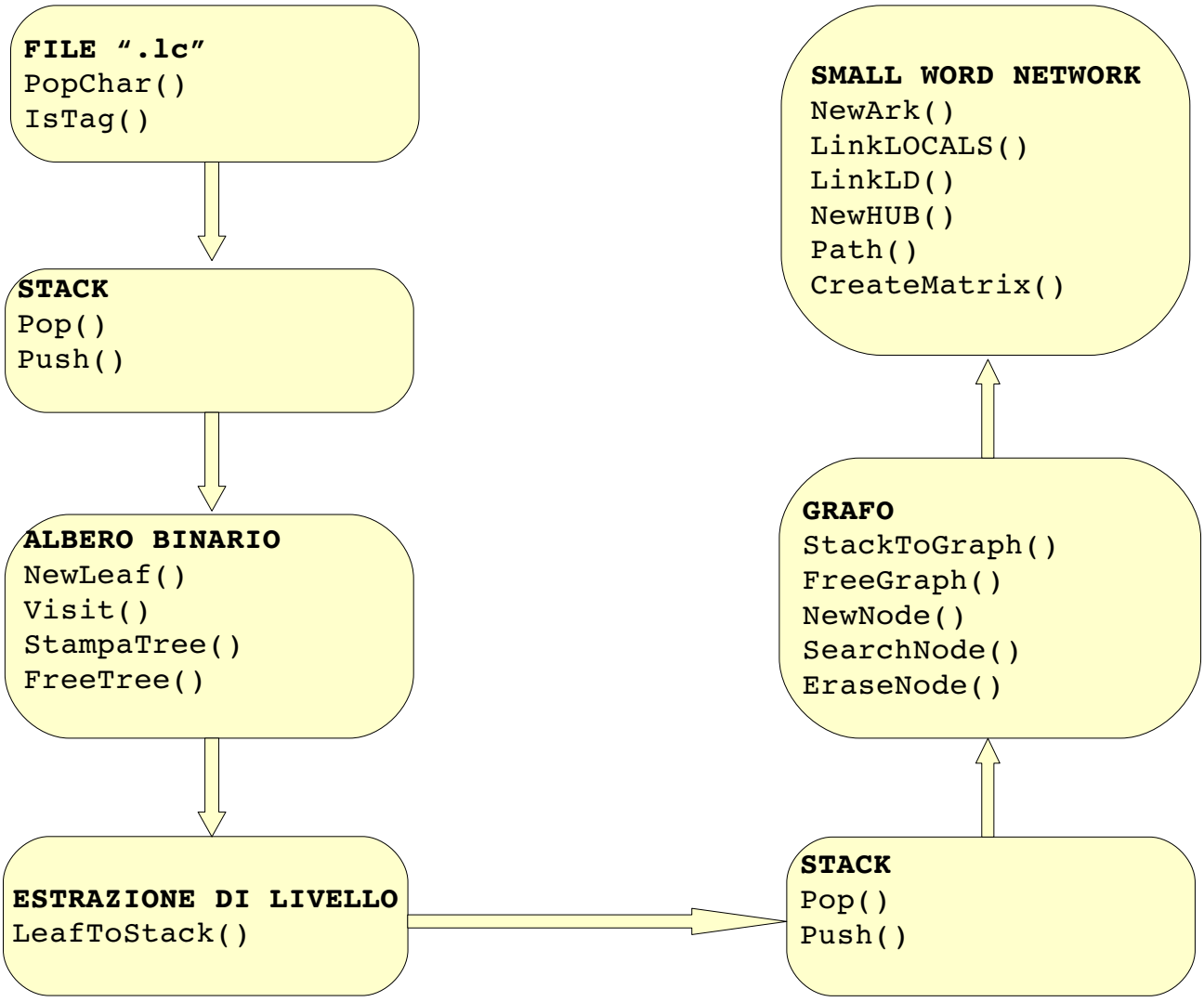
### *1.1.L'applicazione*

Dopo le solite numerose (a volte inconcludenti) ponderazioni riguardo al trovare una possibile applicazione, per rendere meno drastico e più interattivo l'approccio ad un simile progetto, quasi casualmente siamo venuti in possesso di una particolare “quantità di dati” sensati e versatili al nostro scopo:
Una lista contenente in ordine alfabetico tutti i comuni d'Italia con relative coordinate geodetiche (latitudine e longitudine).
Una situazione perfetta per lo sviluppo di una rete piccolo mondo che abbia proprietà specifiche e il più possibile vicine alla realtà.
Con le coordinate è molto semplice calcolare la distanza chilometrica (in linea d'aria) tra due comuni italiani...ergo i pesi degli archi !!!
Per utilizzare al meglio il file pervenutoci ci è bastando convertirlo nell'oramai consueto formato “.lc” (si veda documentazione **pr2gr38**) secondo la sintassi esposta nella sezione dedicata.
Partendo da questi dati ci siamo proposti di realizzare un piccolo linker di distanze aeree, dei comuni italiani (volendo anche di tutto il mondo).
La quantità dati elevata (22.560 comuni) ci ha portato ad ottimizzare al meglio la gestione

delle strutture dati, dei memory leak e gli algoritmi di esportazione visita ed eliminazione. Ma tutto questo lo vedremo meglio nelle specifiche sezioni.

### *1.2.Analisi Implementativa Generale*

Stabilita la generalità dei dati sui quali lavorare, è stato necessario organizzare l'approccio alle varie parti del progetto e come queste ultime dovevano essere relazionate fra loro; non pochi fastidi ha, sotto questo punto di vista, generato l'inserimento della struttura dati "Albero" in modo da non compromettere il fine ultimo del programma, cioè analizzare il comportamento computazionale di una Small Word Network.
Per rendere meno invasiva, nei confronti della memoria la gestione di due strutture dati così grandi è stata molto utile l'implementazione di uno STACK usato per effettuare le transizioni degli elementi dall'albero binario al Grafo.
Lo schema seguente[1] può sostituire inutili elucubrazioni teoriche dalle quali il concetto potrebbe solo risultarne travisato:




*1.Le function inserite sono da considerarsi di tipo generale, e non quelle ufficiali.*

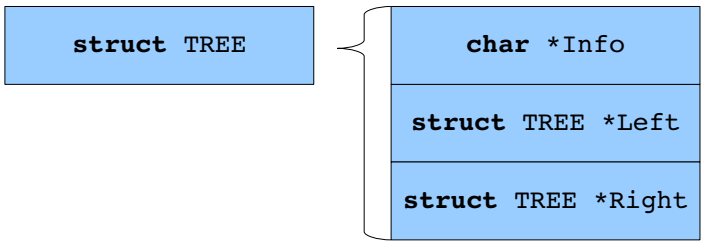
### *1.3.La Struttura Dati*

Analizziamo ora nel dettaglio, le definizioni di tipo e forma delle strutture dati usate.
(i path sono relativi alla cartella dell'eseguibile).

### *1.3.1.L'albero Binario (./include/tree.h)*

La costruzione dell'albero binario avviene, basandosi su file di testo, tramite la seguente forma di struttura dati.

**Modello Generale:**



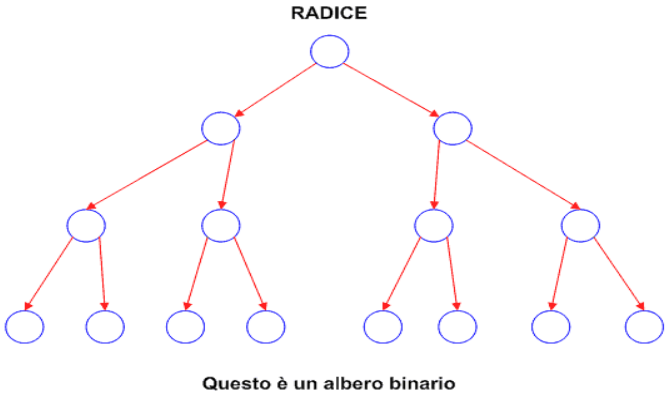

**Modello reale**

```
#ifndef _TREE_STRUCT_H_ //se non definita (si evitano link error)
#define _TREE_STRUCT_H_ //definisce la struttura seguente

#ifndef BUFFSIZE        //buffer di lettura stringhe
#define BUFFSIZE 128
#endif

typedef struct TREE {
                        char  City[BUFFSIZE]; //campo città
                        int   Region;         //regione geografica
                        double Lat;           //latitudine
                        double Long;          //longitudine
                        struct TREE *Left;    //puntatore figlio Sx
                        struct TREE *Right;   //puntatore figlio Dx
                    } TREE;
#endif
```

Non è di difficile comprensione, ogni nodo punta al massimo a due figli che a loro volta (dovendo costruire un albero bilanciato) punteranno ad altri link, che solo nel caso dell'ultimo livello punteranno a NULL.




Questo è un albero binario

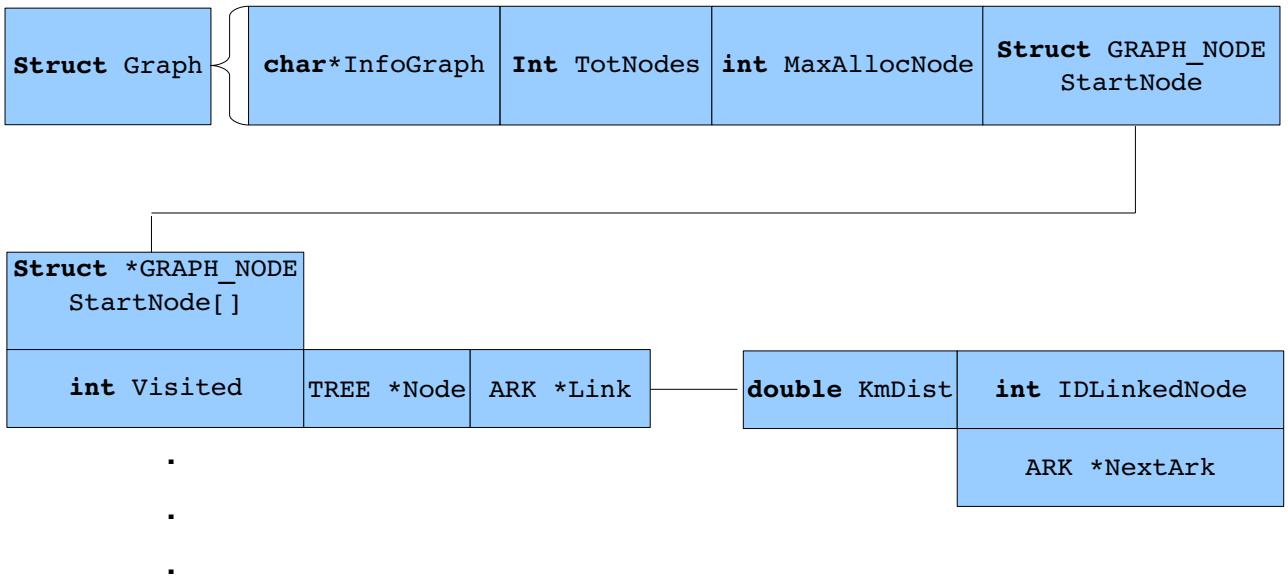
### *1.3.2.Il Grafo(./include/graph.h)*

La reallizzazione della medesima struttura dati ha destato non poche perplessità in merito all'adottare il giusto criterio di rappresentazione.
In linea di principio le scelte sono due:
*1.Matrice di Adiacenza;*
*2.Liste di Adiacenza;*
La prima, semplice da implementare ma di complessità più elevata, permette di avere poche informazioni sul grafo (è difficile l'attribuzione dei pesi agli archi).
Nel caso di grafi Orientati, è costituita da una matrice quadrata i cui indici rappresentano i nodi.
Se fra due nodi esiste il link viene posto un 1 sulla matrice.
È naturale pensare che se si tratta (come nel nostro caso) di un grafo non orientato, la matrice diventa triangolare superiore.
Esempio:

| | | | | | | |
|---|---|---|---|---|---|---|
| **1** | 0 | 0 | 0 | 0 | 0 | >Esiste un collegamento fra i NODI 2 e 4, |
| 1 | **1** | 1 | 1 | 0 | 0 | >**Matrix[2,4]=1** |
| 0 | 0 | **1** | 1 | 0 | 1 | |
| 0 | 0 | 0 | **1** | 0 | 0 | |
| 0 | 1 | 0 | 0 | **1** | 0 | |
| 0 | 0 | 1 | 1 | 0 | **1** | |

La diagonale principale è sempre unitaria, in quanto in ogni grafo tutti gli elementi godono della proprietà riflessiva.
Le liste di adiacenza invece sono un po' più complesse nell'implementazione ma molto più versatili ed economiche in senso computazionale.
Questo è dovuto alla concezione della struttura come puntatori a liste concatenate a differenza della matrice che genera un overhead di memoria piuttosto ampio.
La nostra scelta si è indirizzata verso l'ultima soluzione, di sicuro più elegante ed adatta allo scopo.
Ma vediamo nel dettaglio la definizione della nostra struttura.

**Modello Generale**




**Modello Reale per ordine di definizione**

la struttura ARK (gli archi di un nodo). Una lista concatenata che contiene gli ID dei nodi che costituiscono il collegamento, e il campo dedicato al peso degli archi.

```
struct ARK{                         //ARCO
            double KmDist;          //Lunghezza in KM dell'ARCO (Peso)
            int IDLinkedNode;       //id nodo collegato
            struct ARK *NextArk;//Link al prossimo arco del nodo
          };
typedef struct ARK ARK;
```

Definita la struttura degli archi si può costruire la definizione di nodo di un grafo:

```
struct GRAPH_NODE{                  //NODO
                    TREE *Node; //LINK di tipo TREE(nity)
                    ARK *Ark;   //archi del nodo (LISTA DI ARCHI)
                    int Visited;//flag di NODO visitato
                  };
typedef struct GRAPH_NODE GRAPH_NODE;
```

La scelta di utilizzare come campo “Info”, il tipo pre-definito **TREE** è dovuta ad una facilità di gestione della transizione dei nodi dalla albero binario allo stack; evitando inutili swap di variabili che avrebbero portato solo ad un aumento dei **memory leak** e ad un aumento della complessità di spazio.
Infine viene definito il tipo strutturato Graph, che punta al primo nodo del grafo (quello con ID =0).

```
struct GRAPH{
            char InfoGraph[BUFFSIZE];   //nome grafo
            int TotNodes;               //nodi in totale
            int MaxAllocNode;     //nodi liberi prima del realloc
            GRAPH_NODE *StartNode;//puntatore al primo nodo
            };
typedef struct GRAPH GRAPH;
```

La scelta di utilizzare la variabile intera MaxAllocNode, ci è stata suggerita dall'esempio di grafo del professor L.Lamberti, che come egli stesso ha spiegato è importante per garantire una frammentazione più regolare della memoria. Ciò è inutile nel caso di piccole quantità di dati, ma in casi di grandi allocazioni è conveniente ridurre al minimo l'overhead di memoria (si veda la sezione sui memory leak e il debug).

### *1.3.3.Lo STACK(./lib/stack/stack.h)*

La struttura dati STACK si è rivelata un ottima scelta per quanto riguarda la gestione delle strutture “maggiori”.
Per la sua conformazione è l'ideale per l'implementazione di visite e caricamenti, o controlli sui dati.
La sua definizione è molto semplice:

```
#define _STACK_STRUCT_H_
struct stack{
              TREE  *Item;
              struct stack *Next;
```

```
        };

typedef struct stack STACK;
```

Ovviamente il campo “Info” dello stack è di tipo TREE*,  per l'ottimizzazione di cui sopra.

### *1.4.Il File*

Come detto in precedenza il nostro Albero binario viene caricato a partire da un particolare tipo di file.
Ovviamente il formato di descrizione file **“.lc”** (si veda documentazione **pr2gr38**), è venuto incontro alle nostre esigenze.
La quantità di dati è stata dapprima trasformata con la sintassi lc:

```
<begin>
<class>
     <CITY>Napoli</CITY>
     <REG>04</REG>
     <LAT>23.000034</LAT>
     <LONG>12.00054</LONG>
</class>
</begin>
```

Function già viste in precedenza si occupano della corretta acquisizione dei dati dal file.

## 2.Analisi Implementativa (completa)

### *2.0.La struttura del main (./linkEarth.c)*

Il main, incluso nel file linkEarth.c è strutturato in modo molto semplice:

```
main ()
{
        _dichiarazione delle variabili principali,
        .
        _Inizializzazione Grafo e Stack
        ClearScreen();  //settagio della GUI iniziale;
        GetConf();
        Splash();

do
{       display del menù principale:
        DispBackground();
        DispForeground(MenuMAIN,i);

switch(GetKey())
    {
        viene gestito il movimento all'interno del menù;
        case UP
        ...
        case DOWN
        ...
        case ENTER:
```

```
        ...
        {
                vengono chiamati i menù richiesti;
}
```

Per una comprensione più dettagliata vi rimandiamo ad una lettura della documentazione specifica delle function e del medesimo codice sorgente.

### *2.1.Function per la lettura del file (include/lcfile.h)*

```
id[2x13]void  PopChar (FILE *lc, char **ReadBuffer);
```

Function per la lettura stringa per stringa di un file di testo.
Si arresta nel caso in cui trovi caratteri speciali di inizio o fine tag.
La function è già stata trattata nel precedente progetto.
(documentazione contactManager, PopChar**[id2x03]**).

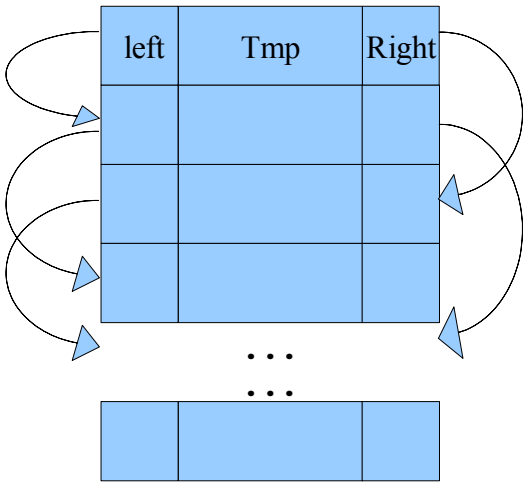
```
id[2x12]int   IsTag   (char *ReadBuffer);
```

La function IsTag verifica se una data stringa è un TAG, e in caso positivo ne restituisce il codice numerico.
Anche questa funzione è un riutilizzo della IsTag del progetto precedente, adattando soltanto i valori ai nuvi tag.

### *2.2.Function per la gestione dell'albero binario (include/libtree.h)*

```
id[3x01]TREE *StackToTree(STACK **Stk, int *c);
```

La funzione legge i valori da uno STACK, e genera un albero binario completamente bilanciato.
Basandosi su di uno stack di appoggio scorre i valori “linkandoli” ad albero come da figura:




La funzione è di tipo TREE, e restituisce l'albero creato (completo).
Man mano che si scorre lo stack e gli elementi vengono collegati fra loro. La memoria occupata dallo stack viene liberata ad ogni prelievo di elementi. Gli elementi non sono copiati in una nuova struttura, ma è solo la loro immagine in memoria che cambia,

assumendo la forma di albero binario.
Per prelevare gli elementi dallo stack viene chiamata la funzione Pop_STACK(Stk)
**[id stackx03].**

id[3x02]
```
int  NewLeaf     (TREE **Tree);
```
La funzione NewLeaf, alloca un nuovo nodo di tipo TREE e ne inizializza i valori.
```
Tmp=(TREE*)malloc(sizeof(TREE));
   if(Tmp!=NULL)
   {
      // inizializza campi
      strcpy(Tmp->City,"\0");
      Tmp->Region=0;
      Tmp->Lat=0;
      Tmp->Long=0;
      // inizializza figli
      Tmp->Right= NULL;
      Tmp->Left = NULL;
```

id[3x03]
```
int  HeightVisit (TREE *Tree, int c);
```
Visita in profondità dell'albero binario.
La funzione (ricorsiva), se il nodo root è diverso da NULL, si autorichiama, per il figlio sinistro e per il figlio destro.
La variabile **int*** c, assume il valore del livello dell'albero.
```
if(Tree!=NULL)
   {
      (*c)++;
      if(Tree->Right!=NULL)
         HeightVisit(Tree->Right,c);
      if(Tree->Left!=NULL)
         HeightVisit(Tree->Left,c);
```
id[3x04]
```
int  LcOpen      (TREE **Tree);
```
Si occupa della gestione della lettura del file ".lc", chiamando le varie funzioni di lettura del file.
Man mano che gli elementi vengono trovati vengono chiamate le funzioni di controllo usuali, per il riconoscimento dei tag, e il controllo degli inserimenti nel buffer.
```
if(lc!=NULL)
   {
      do
         PopChar(lc,&ReadBuffer);
  //i caratteri restituiti da PopChar vengono salvati nel Buffer
      while(IsTag(ReadBuffer)!=BEGIN && !feof(lc));
      Init_STACK(&Stk);
      ReadClass(lc,&Stk);
      fclose(lc);
```
In caso di file corrotto, viene restituito un codice errore. (si veda l'apposita sezione).
Alla fine lo stack carico(restituito da ReadClass), viene passato alla funzione StackToTree, che crea l'albero binario.

id[3x05]
```
int  ReadClass   (FILE *lc, STACK **Stack);
```
Dato il file di testo effettua il push sullo stack ogni qualvolta viene trovato un elemento

<class>.
Uno switch fra i due possibili tag, permette di effettuare l'operazione necessaria.

```
switch(Tag)
   {
      case CLASS : NewLeaf(&Leaf);
                   LoadLeaf(lc,&Leaf);
                   Push_STACK(Leaf,Stack);
                   ReadClass(lc,Stack);
                   break;

      case END   : Error=0; break;
   }
free(ReadBuffer);
```

Alla fine viene liberato il buffer di lettura e torna lo stato di errore (0 in caso di assenza).

```
id[3x06]int  LoadLeaf    (FILE *lc, TREE **Tree);
```

La funzione LoadLeaf, (riutilizzata in parte dalla funzione **Load** del contactManager), effettua lo switch dei tag trovati, e seconda del valore posiziona gli elementi nel campo giusto della struttura TREE.

```
switch(IsTag(ReadBuffer)) // identifica tag contenuto nel buffer
     case CITY :    //effettua le operazioni del caso
// se la parola non supera BUFFSIZE
          if((strlen((*Tree)->City)+strlen(ReadBuffer))<=BUFFSIZE)
          // concatena stringhe eventualmente separate
          // da caratteri non accettati e le carica nel
          // campo della struttura.
          strncat((*Tree)->City,ReadBuffer,BUFFSIZE);
```

E così via per gli altri casi (regione, latitude ecc.).
Come di consueto la funzione alla fine delle operazioni libera i BUFFER allocati e restituisce gli eventuali codici errore.

```
id[3x07]void StampaTree  (TREE *Tree,int h);
```

Stampa dell'albero binario su file di testo.

```
id[3x08]void LeafsToStack(TREE **Tree,STACK **Leafs,int Weight);
```

Function per la ricostruzione dello stack a partire dall'albero binario.
Ripone nello stack gli elementi di un dato livello visitandoli in ampiezza.
Accetta in input l'albero, lo stack, l'altezza e il livello dal quale estrarre gli elementi.

```
Push_STACK(*Tree,Leafs);
      if(*Tree)
      {
         if((*Tree)->Left)
         {
            FreeTree(&(*Tree)->Left);
            (*Tree)->Left=NULL;
         }
         if((*Tree)->Right)
         {
```

```
            FreeTree(&(*Tree)->Right);
            (*Tree)->Right=NULL;
```
Questa funzione man mano che costruisce lo stack libera la memoria occupata dall'albero. Chiamando la funzione **FreeTree**(segue).

```
id[3x09]void FreeTree    (TREE **Root);
```
Funzione che visitando l'albero in profondità, ne libera la memoria allocata:
```
if((*Root)!=NULL)
   {
      FreeTree(&(*Root)->Left);
      FreeTree(&(*Root)->Right);
      free(*Root);
```
Alla fine la memoria è completamente libera.

### *2.3.Function per la gestione del grafo e della rete S.W.N.(include/libgraph.h)*

```
id[3x10]void NewArk      (GRAPH *CityGraph, int ID_A, int ID_B);
```
Alloca un nuovo elemento di tipo ARK, da aggiungere ad un nodo del grafo al quale è stato assegnato un altro arco.
Accetta in input il Grafo e i due id degli elementi da collegare.
Viene effettuato lo scorrimento delle linked list dei relativi nodi.
Viene allocato l'elemento e lo si fa puntare alla struttura degli archi.
La funzione chiamata una sola volta crea un arco orientato, è quindi sufficiente richiamarla due volte invertendo i valori degli ID.
In questo modo generiamo un grafo non orientato, ma comunque generale e bene predisposto a problemi di natura orientata.

```
id[3x11]void NewNode(GRAPH *CityGraph, int *MaxAllocNode, int
Size);
```
Alloca BATCH_NODE elementi al grafo se le posizioni libere sono finite.
```
(GRAPH_NODE*)
realloc(CityGraph->StartNode,BATCH_NODE+Size*sizeof(GRAPH_NODE));
```

Se le posizioni libere non sono finite invece di reallocare la funzione decrementa la variabile MaxAllocNode all'interno della struttura Graph.

```
id[3x12]void StackToGraph (STACK **Leafs,    GRAPH *CityGraph);
```
Dallo stack creato dalla funzione LeafsToStack, viene costruito il grafo.
Viene allocata la struttura GRAPH_NODE, che alla fine conterrà il grafo completo ma ancora sconnesso.
I puntatori alle strutture ARK vengono impostati a NULL, per evitare successivi problemi di allocazione e di lettura degli ARCHI.
Come ultima azione la funzione chiama LinkLOCALS, che collegherà tutto il grafo ad “anello”, basandosi sulle topologie di rete per l'appunto di tipo “ring”.

```
id[3x13]void LinkLOCALS   (GRAPH *CityGraph, int TotNode);
```
La funzione LinkLOCALS crea, come detto in precedenza, un prima forma di rete SWN, collegando i nodi ad anello.
Il funzionamento è molto semplice:
Vengono allocati gli archi e in ogni campo IDLinkedNode della struttura viene assegnato

l'id successivo,
Inoltre vengono attribuiti i pesi agli archi (nel campo KmDist), ricordiamo che i pesi sono la distanza geodetica fra i nodi, e che nella gestione delle rete assumono un ruolo di secondaria importanza. In quanto le distanze vengono intese come il numero di  nodi attraversati.
Il Peso ci è utile per questioni di scelta legate al caso e per garantire una migliore interazione con l'utente.

id[3x14]`int *SearchNode(GRAPH CityGraph, char *Key, int Size, int *IDAlloc);`
Data in input una chiave di ricerca, di tipo char, viene effettuata la ricerca dei nodi per nome.
La funzione restituisce un array contenente gli ID trovati (in caso di omonimi).
La variabile *IDAlloc è per l'appunto la dimensione dell'array.

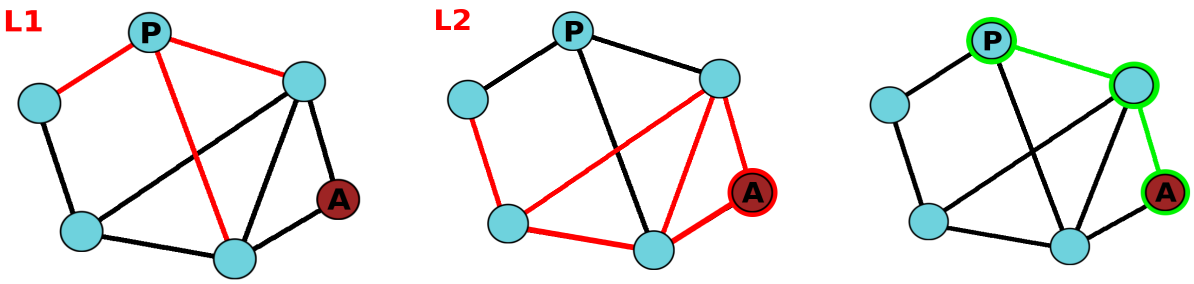
id[3x15]`void FreeGraph    (GRAPH *Graph);`
Funzione di liberazione della memoria occupata dal grafo.
Dapprima vengono deallocate le strutture degli archi e, man mano vengono chiamati i free sui nodi.

```
for(i=0;i<AlNodes;i++)
   {
      while(Curr[i].Ark!=NULL)
      {
         TmpArk=Curr[i].Ark;
         Curr[i].Ark=TmpArk->NextArk;
         free(TmpArk);
      }
      if(Curr[i].Node!=NULL)
         free(Curr[i].Node);
   }
   free(Graph->StartNode);
   Graph->StartNode=NULL;
```

Alla fine il grafo scompare completamente dalla memoria.

id[3x16]`void EraseNode    (GRAPH *CityGraph, int ID);`
Funzione di cancellazione di un elemento dato l'ID.
Si posizione nel nodo indicato dall'ID; qui scorre la lista degli archi salvando gli ID in un array.
Svuotata la struttura e deallocata, il campo viene fatto puntare a NULL, è quindi disponibile per scritture successive.
Partendo dal primo elemento dell'array (contenente l'ID del primo link del nodo eliminato), ci si posiziona in tutti i nodi che erano collegati all'eliminato.
La lista degli archi di ogni nodo viene fatta scorrere con delle liste di supporto in quanto è necessario non perdere gli elementi.
Deallocato il nodo contenente il link cercato la struttura viene riassegnata al nodo del grafo.
Questa operazione viene effettuata per ogni ID dell'array.

id[3x17]**void** Path (GRAPH *CityGraph, **int** **Line, **int** Dim, **int** *Counter, **int** ID);
Cerca il percorso più breve tra due nodi all'interno del grafo.
La ricerca viene effettuata tramite l'esplorazione di un albero n-ario generato dal grafo. L'elemento di **root** è costituito dal nodo di partenza. I suoi figli sono i nodi puntati dai suoi archi. In questo modo, ogni livello dell'albero è costituito dagli archi dei nodi del livello precedente. La function è quindi costituita da un blocco iterativo ed uno ricorsivo. Il blocco iterativo, analizza gli archi dei nodi del livello attuale (es. primo livello, archi del nodo di partenza. Secondo livello, archi dei nodi puntati dal nodo di partenza ecc...). Gli ID dei nodi puntati dagli archi vengono salvati in un array (NewLine), che costituirà il nuovo livello dell'albero. Se tra gli archi, ne esiste uno che punta al nodo di arrivo, l'iterazione si interrompe e una flag (Trovato) viene asserita. Il blocco ricorsivo verifica lo stato della flag "Trovato" e, nel caso in cui non sia asserita, richiama la function per il nuovo livello appena generato. In caso contrario, genera un array di dimensioni pari al numero di passi effettuati, e lo restituisce alla function chiamante. E' facile intuire che ogni chiamata ricorsiva della function Path, costituisce un "passo" verso il nodo di arrivo. L'array di output viene riempito con gli id dei nodi che costituiscono il percorso più breve.




id[3x18]**void** CreateMatrix (GRAPH *CityGraph, **int** ***Matrix);
Crea la matrice di adiacenza partendo dalla struttura dati primaria.

id[3x19]**void** LinkLD (GRAPH *CityGraph, **int** FarNodes);
Function per l'inserimento di collegamenti LD (Long Distance).
Accetta in input il numero di nodi che dovranno avere dei collegamenti LD.
Viene quindi dichiarato uno step:

```
Step=(CityGraph->TotNodes)/FarNodes;
```

questa sarà sempre la distanza fra due nodi aventi LinkLD.

```
for(i=0;i<=(CityGraph->TotNodes)/2;i+=Step)
{
   NewArk(CityGraph,i,
          ((CityGraph->TotNodes/2)+i)%CityGraph->TotNodes);
   NewArk(CityGraph,
          ((CityGraph->TotNodes/2)+i)%CityGraph->TotNodes,i);
}
```

In questo modo colleghiamo i nodi a distanza TotNodes/2 fra loro, in modo da avere una

rete più omogenea sulla quale procedere in seguito con le sperimentazioni computazionali.

id[3x20]**void** NewHUB (GRAPH *CityGraph, **int** ID, **int** LinkRank);
La funzione permette di assegnare ad un Nodo la peculiarità di essere un HUB.
**HUB =** grande quantità di link LD e Locals.
Accetta in input l'ID del nodo da trasformare e il suo LinkRank, cioè il numero di link che deve possedere.
Anche in questa funzione viene dichiarato uno step, che questa volta corrisponde alla distanza fra gli arrivi dei link che partono dallHUB.

```
Step=(CityGraph->TotNodes)/LinkRank;
for(i=0;i<(CityGraph->TotNodes);i+=Step)
{
   if(i!=ID)
   {
      NewArk(CityGraph,ID,i);
      NewArk(CityGraph,i,ID);
   }
```

id[3x21]**float** HUBRank (GRAPH *CityGraph, **int** ID); **[deprecated]**
Funzione per la scelta di un HUB in base al peso.
Se gli elementi sono collegati secondo lo schema iniziale, l'HUB viene selezionato in base al Peso medio massimo rispetto agli altri nodi, sancendone così il Rank che lo candida ad essere scelto come HUB.

id[3x22]**double** GeoDist (**double** LatA, **double** LonA, **double** LatB, **double** LonB);

Funzione per il calcolo della distanza geodetica.
Si basa sulle comuni funzioni trigonometriche per il calcolo di distanze sferiche.

### *2.4.Function per la gestione dello STACK(lib/stack/libstack.h)*

id[stackx01]**void** Init_STACK (STACK** );
Crea ed Inizializza lo STACK.

id[staclx02]**void** Push_STACK (TREE*, STACK**);
Effettua il Push sullo STACK.

id[stackx03]**TREE*** Pop_STACK (STACK** );
Effettua il Pop sullo STACK.

id[stackx04]**void** Free_STACK (STACK** );
Libera la memoria occupata dalla struttura dati stack.

id[stackx05]**boolean** Empty_STACK(STACK** );
Verifica se lo stack è vuoto.

### *2.5.Function costituenti la GUI(include/GUI.h)*

Segue l'elenco delle funzioni costituenti la pseudo-GUI (graphic-user-interface) per l'interazione con l'utente.
È stata realizzata totalmente in linguaccio C-ANSI, a partire dal framework Platform (visto nei precedenti progetti).
Le limitazioni interattive se presenti, sono da considerarsi limiti del linguaggio C medesimo e, in generale, delle applicazioni eseguite in ambienti BaSh (Unix-Shell) o CMD (Win32-Console).

**(files://./GUI/)**

```
id[3x25]void Splash          ();
```

Function per la stampa a video dello SplashScreen, all'avvio del programma.

```
id[3x26]void DispBackground ();
```

Funzione per il display del background.
Printf orientati e colorati disegnano lo sfondo del programma.

```
id[3x27]void _DispBackground();
```

Identica alla funzione precedente, ma realizzata con fprintf, in questo modo aumenta la priorità della stampa. (importante in caso di utilizzo in funzioni ricorsive);

```
id[3x28]void DispPopup       ();
```

Funzione per la costruzione e la notifica degli elementi di tipo PopUp. (notifiche, inserimenti ecc.);

```
id[3x29]void DispForeground (int MenuID, int i);
```

In base al'ID che gli viene passato per valore, stampa il menù richiesto.

```
id[3x30]void DispTree        (TREE *Root, int i, int j, int Level);
```

```
id[3x31]void DispGraph       (GRAPH *Grafo, int i, int j, int
ArkNum);
```

Gestione della stampa a video del Grafo. (visualizzazione degli archi, scorrimento)

```
id[3x32]void DispBar(int x,int y, int Max, int Actual)
[deprecated]
```

**(files://./Gestione/)**

```
id[3x33]void MenuTree        (STACK**);
```

Menù che gestisce le informazioni e le azioni da eseguire sull'albero.

```
id[3x34]void MenuTree_View  (TREE* Root,TREE* Tree,int i,int j,int
Level,int Begin,int*End);
```

Menù che integra le funzioni di scorrimento e visualizzazione dell'albero binario

```
id[3x35]void MenuTree_Level (TREE* Tree, int Height, int *Level);
```

Menù che gestisce le azioni riguardanti l'estrazione di livello da eseguire sull'albero.

id[3x36]**void** MenuGraph (GRAPH *Grafo);
Menù contenente le azioni riguardanti il grafo e la sua gestione.

id[3x37]**void** MenuGraph_View (GRAPH *Grafo);
Menù di gestione delle operazioni da effettuare durante la stampa a video del Grafo.

id[3x38]**void** MenuGraph_Mod (GRAPH *Grafo);
Menù per visualizzare le azioni di modifica del grafo.
id[3x39]**void** MenuGraph_Mod_LD (GRAPH *Grafo);
Menù di aggiunta archi LD.

id[3x40]**void** MenuGraph_Mod_AddNode (GRAPH *Grafo);
Menù aggiungi nodo.

id[3x41]**void** MenuGraph_Mod_Ark (GRAPH *Grafo);
Menù aggiungi arco.

id[3x42]**void** MenuGraph_Mod_Hub (GRAPH *Grafo);
Menù aggiungi HUB alla rete.

id[3x43]**void** MenuGraph_Mod_Delete (GRAPH *Grafo);
Menù eliminazione nodo del Grafo.

id[3x44]**void** MenuGraph_Path (GRAPH *Grafo);
Menù di ricerca dei percorsi minimi.

id[3x45]**void** MenuGraph_Path_View (GRAPH *Grafo,**int***Steps,**int** Dim);
Menù per la visualizzazione dell'output della function Path.

id[3x46]**void** MenuOption ();
Menù OPZIONI. (cambia visualizzazione)

id[3x23]**void** GetConf ();
Carica file di configurazione.

id[3x24]**void** SetConf ();
Setta configurazione.

id[2x15]**char** *GetchInLine (**int** c,**int** line);
Function per l'acquisizione dei caratteri nei campi di inserimento.
**Riutilizzata dal progetto precedente.**

id[2x16]**char** *GetNumInLine (**int** c,**int** line);
Function per l'acquisizione dei numeri nei campi di inserimento.
**Riutilizzata dal progetto precedente.**

### *2.6.Libreria Platform(lib/platform/platform.h)*
*Framework* per la programmazione multi-piattaforma.

Già analizzato negli elaborati precedenti; è comunque fornita una sua documentazione interna nella cartella di origine dei sorgenti.

### *2.7.Riutilizzo del software*

Come visto in precedenza alcune funzioni fanno parte di progetti precedenti.
Sono state implementate in modo semplice e veloce, perché precedentemente pensate per una rapida esportazione.
Anche funzioni non dichiarate come riutilizzate sono state talvolta riprese da vecchie funzioni adatte ai vari scopi.
Questo tipo di approccio ci ha permesso di lavorare meglio su aspetti più importanti e formativi

## 3.Analisi Computazionale

### *3.0.Analisi sperimentale*

La nostra analisi sperimentale si è soffermata in modo particolare sulla funzione Path, il calcolo della distanza (in nodi), dati due id.
Una prima versione ha comportato non pochi problemi, ma dopo numerosi tentativi e "ristesure" , abbiamo ottenuto una versione dalla complessità soddisfacente e dai tempi di esecuzione piuttosto bassi.
L'analisi è stata effettuata a partire dal grafo estratto con il penultimo livello dell'albero binario (nel nostro caso il 13° livello = 8192 NODI).
I test seguenti si basano sui comportamenti delle rete e della funzione in base ai vari tipi di configurazione (vedi legenda).

| PATH | TEST_1 | TEST_2 | TEST_3 | TEST_4 | TEST_5 | TEST_6 |
|---|---|---|---|---|---|---|
| **12 →79** | 67 steps [331 loops] | 5 steps [866 loops] | 3 steps [2098 loops] | 5 steps [1674 loops] | 3 steps [2804 loops] | 3 steps [4141 loops] |
| **55 →229** | 174 steps [866 loops] | 8 steps [918 loops] | 4 steps [2172 loops] | 3 steps [1743 loops] | 3 steps [2929 loops] | 3 steps [4969 loops] |
| **155 →1000** | 845 steps [4221 loops] | 7 steps [126 loops] | 3 steps [261 loops] | 2 steps [209 loops] | 4 steps [3573 loops] | 2 steps [210 loops] |
| **345 →2346** | 2001 steps [10001 loops] | 11 steos [7449 loops] | 5 steps [6738 loops] | 3 steps [2588 loops] | 2 steps [791 loops] | 3 steps [2586 loops] |
| **778 →5632** | 759 steps [3792 loops] | 6 steps [4831 loops] | 4 steps [1420 loops] | 6 steps [6053 loops] | 4 steps [4022 loops] | 2 steps [2822 loops] |
| **1243→8** | 1235 steps [6173 loops] | 7 steps [2898 loops] | 3 steps [14 loops] | 6 steps [4934 loops] | 4 steps [2756 loops] | 3 steps [17 loops] |
| **7654→3089** | 470 steps [2349 loops] | 6 steps [1774 loops] | 5 steps [3609 loops] | 4 steps [2891 loops] | 4 steps [5322 loops] | 3 steps [6159 loops] |
| **123 →5555** | 1337 steps [6682 loops] | 7 steps [3475 loops] | 4 steps [4835 loops] | 4 steps [1123 loops] | 3 steps [6878 loops] | 4 steps [4661 loops] |
| **6621→8005** | 1384 steps [6916 loops] | 5 steps [4447 loops] | 4 steps [6058 loops] | 3 steps [1634 loops] | 3 steps [8101 loops] | 3 steps [4904 loops] |
| **0 →4444** | 349 steps [1742 loops] | 5 steps [5441 loops] | 2 steps [1119 loops] | 3 steps [4176 loops] | 3 steps [3031 loops] | 2 steps [2230 loops] |
| **5672→777** | 800 steps [3999 loops] | 7 steps [5571 loops] | 3 steps [2449 loops] | 6 steps [6720 loops] | 3 steps [272 loops] | 3 steps [4494 loops] |
| **55 →1423** | 1384 steps [6736 loops] | 10 steps [5771 loops] | 4 steps [2771 loops] | 4 steps [5493 loops] | 4 steps [3925 loops] | 4 steps [6844 loops] |
| **136 →6783** | 1545 steps [7723 loops] | 9 steps [6838 loops] | 3 steps [5448 loops] | 5 steps [7635 loops] | 3 steps [2272 loops] | 3 steps [7632 loops] |
| **3109→4002** | 893 steps [4461 loops] | 5 steps [4484 loops] | 5 steps [7152 loops] | 5 steps [5726 loops] | 3 steps [1347 loops] | 3 steps [2011 loops] |
| **1278→7945** | 1525 steps [7623 loops] | 6 steps [4430 loops] | 5 steps [6030 loops] | 4 steps [1629 loops] | 3 steps [8071 loops] | 3 steps [7352 loops] |
| **MEDIE e INFO** | | | | | | |
| **STEP** | 984,53 steps | 6,93 steps | 3,7 steps | 4,2 steps | 3,2 steps | 2,9 steps |
| **Recursive LOOPS** | 4907,67 loops | 3954,6 loops | 3478,27 loops | 3615,4 loops | 3739,6 loops | 4068,8 loops |
| **Tot Links** | 8600 | 9419 | 10238 | 11466 | 13107 | 12696 |
| **Overhead** | 1643,2 | 14,3 | 8,7 | 9,5 | 7 | 5,8 |

**Legenda:**
Di seguito spieghiamo i vari test e il tipo di variabili indicate.
Step → costo del percorso in passi.
Loop → costo in cicli ricorsivo-iterativi dell'algoritmo.
Tot Links → numero di link attivi nella rete al momento del test.

Overhead→ per determinare l'overhead di una data configurazione di rete, utilizziamo la seguente formula:
**OVERHEAD = [NODI(8191)*Media(STEP)]/RecLOOPS ;**

Questo perchè i cicli non sono altro che i percorsi esplorati nel grafo, di conseguenza il rapporto fra il costo medio del grafo e i percorsi che l'algoritmo visita (complessità), fornisce un discreto indice di funzionalità della rete:

**OVERHEAD → 0 [rete ottimizzata]**
**OVERHEAD → ∞[rete non funzionale]**
Segue la spiegazione dei test effettuati.

**TEST_1**
Solo link L.D. Al **5%(~409 links)** di copertura totale.
Il primo test, come da richiesta del committente, viene effettuato con la sola presenza di nodi LD, presenti in quantita abbastanza limitata.
La scarsità dei percorsi obbligano la funzione Path ad esplorare quasi tutte le possibili vie, provocando un alto overhead.
Il carico della rete è ben distribuito, ma la media degli step è piuttosto alta.
I nodi LD, non sono una buona soluzione per la gestione di problematiche di questo genere di topologie.
C'è da dire pero' che una rete con alla base un'infrastruttura del genere garantisce un'alta tolleranza ai guasti dei nodi; ma vediamo meglio i prossimi comportamenti.

**TEST_2**
Link Long Distance con un rank del **5%(~409 links),** più un HUB all'**id[4096]** con un rank del **10%(~819 links)**.
Questa che vediamo è un'anteprima di quella che potrebbe essere un'ottima configurazione.
Un HUB è in grado di gestire velocemente le connessioni fra i nodi, dislocando collegamenti in modo regolare fra tutti i nodi.
La stessa funzione di ricerca è sottoposta ad un minor numero di esplorazioni.
Inoltre la rete in caso di **"interdizione dell'HUB"**, non diventa totalmente isolata, in quanto la configurazione LD di base fornisce un minimo di linking fra i nodi.
Abbiamo quindi un nuovo parametro da valutare: la **fault-tollerance** di una Small Word Network.

**TEST_3**
Un solo HUB all'**id[4096]**, con con un rank di copertura del **25%(~2047 links)**.
In questo test tutta le comunicazione viene gestita da un grande HUB. La complessità d'esplorazione diminuisce (lievemente), e anche gli step medi, ma il sistema da noi creato pure avendo un overhead accettabile, rimane poco fault-tollerant: se l'HUB è interdetto per qualsiasi motivo, la rete rimane solo con i locals di base, e magari con delle parti completamente isolate.

**TEST_4**
Tre HUB “dislocated”, (**id[2730]-10%, id[5461]-20%, id[8191]-10%**), uniti da una dorsale di collegamento.
Nel quarto test optiamo per una strategia di distribuzione degli HUB, unendoli con una dorsale principale.
Questa topologia ha overhead maggiore, ma è pur vero che è robusta e molto tollerante agli inconvenienti.
La disposizione degli HUB, permette anche in assenza di uno dei tre di sopperire in modo ragionevole alle richieste di collegamento.
La complessità è maggiore, in quanto i percorsi aumentano insieme al numero dei link inseriti, ma tutto sommato è una situazione accettabile.

**TEST_5**
Copertura L.D. Del **15% (~1229 links)**, più due HUB, uno all'**id[4096]-30% (2458 links)**, e uno all'**id[6144]-15% (1229 links)**.
Con il quinto test abbiamo forse l'ottimizzazione migliore:
overhead accettabile, complessità limitata e una media delle distanze molto bassa rispetto al numero dei nodi.
Inoltre una copertura di base LD e l'aggiunta di un HUB di supporto garantiscono la fault tollerance di questo esempio.

**TEST_6**
Due HUB: **id[4096]-35% (2867 links)** e **id[6144]-20% (1638 links).**
Nell'ultima topologia, viene ridotta al minimo la media degli step (2,8), questo garantisce velocità e complessità limitata, nonostante il numero di link contenuto. Ma è proprio il numero di link piuttosto basso (rispetto alle peculiarità di linking della rete) che ne fanno una topologia poco resistente alle interdizioni.

**In generale,** una soluzione vantaggiosa consta di un buon compromesso fra overhead e fault-tollerance.
Nodi di tipo HUB sono più efficaci se posti sulle fondamenta di una rete con un certo numero di link LD e, soprattutto, meglio se dislocati in modo uniforme lungo la S.W.N. In questione.

***3.1.Analisi della function Path***
In linea di massima, questo tipo di visita sui grafi, potrebbe essere effettuato a partire da algoritmi di **BFS (Breadth-First-Search) parziali**, ma in questo caso aumenterebbero solo la complessità.
Volendo analizzare l'algoritmo BFS, la sua complessità di tempo si ricaverebbe in questo modo:
detto **b** il **branching factor** (il numero di figli di un dato nodo), e **d** la profondità massima del grafo è chiaro che ci troveremmo di fronte alla seguente formula:

$$b^1{}_1+b^2{}_2+...+b_n{}^d \text{ (con n=nodi totali),}$$

il cui andamento asintotico è facilmente deducibile: **O (b^d) !!!**
Possiamo dire che non è proprio quello di cui abbiamo bisogno in questa situazione.
Come si evince dai precedenti test, la funzione Path, effettua al più Nodi Totali***step medi** cicli di cui soltanto step medi ricorsioni.
Ci troviamo quindi fuori da livelli esponenziali ma nell'ordine di **O(b*N),** considerando b

com e la media del numero dei figli di ogni nodo**.**

## 4.Debug e Memory Leak

Il programma è stato soggetto a rigidi controlli di debug con i seguenti strumenti:
**gdb;**
**valgrind;**
**gprof;**

Con gdb è stato effettuato il debug di sviluppo, la correzione di errori di **segmentation fault** e l'ottimizzazione della complessità degli algoritmi.
Con valgrind abbiamo ottimizzato la gestione della memoria fisica eliminando tutti i *memory leaks*.
È stato un ottimo supporto per sviluppo di tutte le funzioni di liberazione della memoria.
Gprof è un ottimo strumento di analisi del software che permette di individuare tutte le chiamate di sistema, i jump, le allocazioni e altre azioni che effettua un programma e quanto tempo ogni funzione impiega a concludersi.
Alleghiamo di seguito gli output dei suddetti programmi:
Analisi dei registri di **gdb:**

| | | | |
|---|---|---|---|
| **eax** | **0x0** | **0** | **//memori riservata (NULL=liberata)** |
| **ecx** | **0xbf95dd44** | **-1080697532** | |
| **edx** | **0x0** | **0** | |
| **ebx** | **0xb8004ff4** | **-1207939084** | |
| **esp** | **0xbf95de10** | **0xbf95de10** | |
| **ebp** | **0xbf95de48** | **0xbf95de48** | |
| **esi** | **0x80503f0** | **134546416** | |
| **edi** | **0x8048c60** | **134515808** | |
| **eip** | **0x8048f2f** | **0x8048f2f <main+459>** | **//puntatore alla base dello stack** |
| **eflags** | **0x282** | **[ SF IF ]** | |
| **cs** | **0x73** | **115** | |
| **ss** | **0x7b** | **123** | |
| **ds** | **0x7b** | **123** | |
| **es** | **0x7b** | **123** | |
| **fs** | **0x0** | **0** | |
| **gs** | **0x33** | **51** | |

Possiamo notare che immediatamente prima del termine del programma, solo il main è memorizzato nel puntatore alla base dello stack del processo (**EIP**).
Ergo tutte le funzioni sono terminate regolarmente senza forzature errori, o altri segnali di **eccezioni** o **interrupt**.
Vediamo ora l'ouput di valgrind con chek della memoria abilitato (--leak-check=yes):

Tutta la memoria allocata è stata liberata al termine del programma, nessun byte di memoria è perso.

```
==26073== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 13 from 1)
--26073--
--26073--   supp:     13 dl-hack3-1
```

==26073== malloc/free: in use at exit: 0 bytes in 0 blocks.
==26073== malloc/free: 82,769 allocs, 82,769 frees, 8,644,072 bytes allocated.
==26073==
==26073== All heap blocks were freed -- no leaks are possible.

Come ultima analisi mostriamo il call graph di **gprof.**
Questo è il grafico delle chiamate, mostra il numero di chiamate per function:

| time | seconds | seconds | calls | Ts/call | Ts/call | name |
|---|---|---|---|---|---|---|
| 0.00 | 0.00 | 0.00 | 411035 | 0.00 | 0.00 | PopChar |
| 0.00 | 0.00 | 0.00 | 308269 | 0.00 | 0.00 | IsTag |
| 0.00 | 0.00 | 0.00 | 40946 | 0.00 | 0.00 | NewArk |
| 0.00 | 0.00 | 0.00 | 40934 | 0.00 | 0.00 | GeoDist |
| 0.00 | 0.00 | 0.00 | 28743 | 0.00 | 0.00 | Pop_STACK |
| 0.00 | 0.00 | 0.00 | 28743 | 0.00 | 0.00 | Push_STACK |
| 0.00 | 0.00 | 0.00 | 20551 | 0.00 | 0.00 | LoadLeaf |
| 0.00 | 0.00 | 0.00 | 20551 | 0.00 | 0.00 | NewLeaf |
| 0.00 | 0.00 | 0.00 | 11865 | 0.00 | 0.00 | Gotoxy |
| 0.00 | 0.00 | 0.00 | 9421 | 0.00 | 0.00 | SetColor |
| 0.00 | 0.00 | 0.00 | 8192 | 0.00 | 0.00 | NewNode |
| 0.00 | 0.00 | 0.00 | 4734 | 0.00 | 0.00 | TextReset |
| 0.00 | 0.00 | 0.00 | 4668 | 0.00 | 0.00 | _Gotoxy |
| 0.00 | 0.00 | 0.00 | 4168 | 0.00 | 0.00 | FreeTree |
| 0.00 | 0.00 | 0.00 | 3978 | 0.00 | 0.00 | CPrintf |
| 0.00 | 0.00 | 0.00 | 385 | 0.00 | 0.00 | Getch |
| 0.00 | 0.00 | 0.00 | 384 | 0.00 | 0.00 | GetKey |
| 0.00 | 0.00 | 0.00 | 221 | 0.00 | 0.00 | DispPopup |
| 0.00 | 0.00 | 0.00 | 187 | 0.00 | 0.00 | DispForeground |
| 0.00 | 0.00 | 0.00 | 120 | 0.00 | 0.00 | _SetColor |
| 0.00 | 0.00 | 0.00 | 93 | 0.00 | 0.00 | DispBackground |
| 0.00 | 0.00 | 0.00 | 85 | 0.00 | 0.00 | DispGraph |
| 0.00 | 0.00 | 0.00 | 80 | 0.00 | 0.00 | _TextReset |
| 0.00 | 0.00 | 0.00 | 40 | 0.00 | 0.00 | DispTree |
| 0.00 | 0.00 | 0.00 | 40 | 0.00 | 0.00 | _DispBackground |
| 0.00 | 0.00 | 0.00 | 7 | 0.00 | 0.00 | MenuGraph_Mod_Delete |
| 0.00 | 0.00 | 0.00 | 6 | 0.00 | 0.00 | EraseNode |
| 0.00 | 0.00 | 0.00 | 6 | 0.00 | 0.00 | GetchInLine |
| 0.00 | 0.00 | 0.00 | 3 | 0.00 | 0.00 | GetNumInLine |
| 0.00 | 0.00 | 0.00 | 3 | 0.00 | 0.00 | SearchNode |
| 0.00 | 0.00 | 0.00 | 2 | 0.00 | 0.00 | ClearScreen |
| 0.00 | 0.00 | 0.00 | 2 | 0.00 | 0.00 | MenuGraph_Mod |
| 0.00 | 0.00 | 0.00 | 2 | 0.00 | 0.00 | MenuGraph_View |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | FreeGraph |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | Free_STACK |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | GetConf |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | HeightVisit |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | Init_STACK |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | LcOpen |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | LeafsToStack |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | LinkLD |

| | | | | | | |
|---|---|---|---|---|---|---|
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | LinkLOCALS |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | MenuGraph |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | MenuGraph_Mod_AddNode |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | MenuGraph_Mod_Hub |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | MenuGraph_Mod_LD |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | MenuGraph_Path |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | MenuGraph_Path_View |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | MenuOption |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | MenuTree |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | MenuTree_Level |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | MenuTree_View |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | NewHUB |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | Path |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | ReadClass |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | SetConf |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | Splash |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | StackToGraph |
| 0.00 | 0.00 | 0.00 | 1 | 0.00 | 0.00 | StackToTree |

In base a questo grafico capiamo come vengono gestite le chiamate.
L'altro grafico di output non lo includiamo in quanto molto lungo e dispersivo. Fornisce l'analisi dettagliata delle sottochiamate di ogni funzione.

# 5.Informazioni generali d'uso

### *5.0.Guida alla compilazione Multipiattaforma*

Come già visto nei precedenti progetti, il sorgente sviluppato è completamente compatibile con la piattaforma compilante (vedi documentazione Platform).

#### *5.0.1.GNU/Linux*

Sui sistemi operativi GNU/Linux, aprire il terminale:

```
#estrazione dell'archivio
utente@LINUX~$ tar xvf pr3gr38.zip

#directory dei sorgenti
utente@LINUX~$ cd linkEarth_1.0

#attribuiamo i permessi di esecuzione allo script
utente@LINUX~$ chmod 777 automake.sh

#compilazione
utente@LINUX~$ bash automake.sh start

#per eseguire il programma:
utente@LINUX~$./linkEarth
```

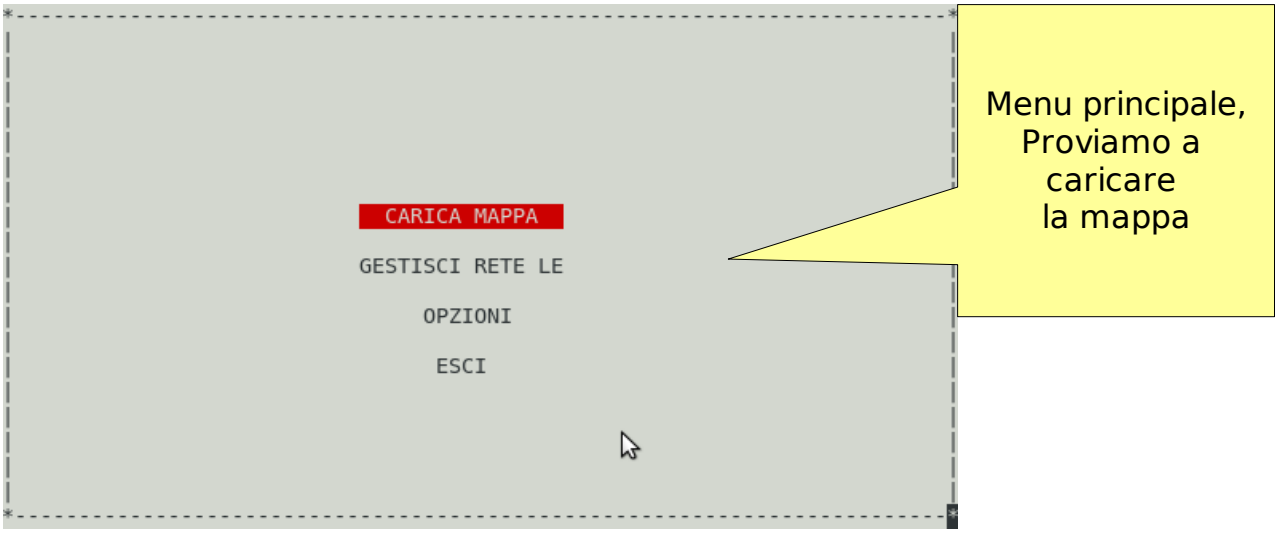
#### *5.0.2.MS-Windows (9x-NT-XP-Vista)*

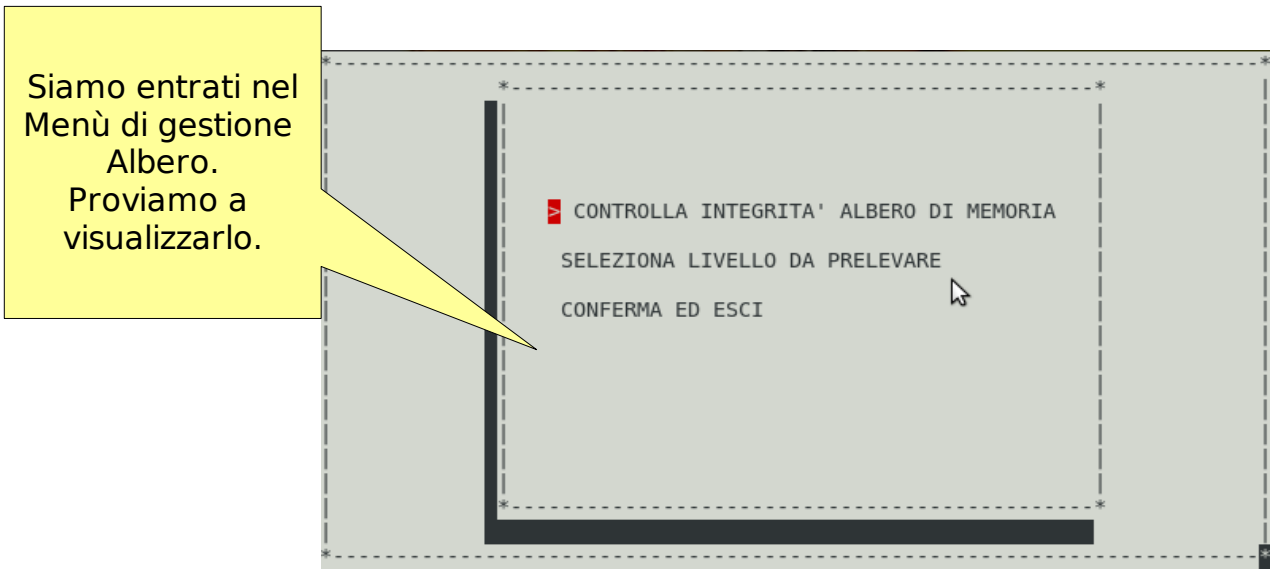
Nei sistemi MS-Windows (tutte le piattaforme), spostarsi nella cartella **C:path...\ProgettoDEV\,** qui ci sono i file di progetto WxDevC++, che permettono la compilazione dei sorgenti.
Per eseguire il file, spostarsi nella cartella **C:path...\ProgettoDEV\Output\MingW\,** e lanciare l'eseguibile **linkEarth.exe**.
Si raccomanda di **NON editare**, i file di configurazioni (Linux.conf, Windows.conf), presenti nelle rispettive cartelle degli eseguibili, onde evitare la perdita delle opzioni di visualizzazione salvate in un precedente utilizzo.

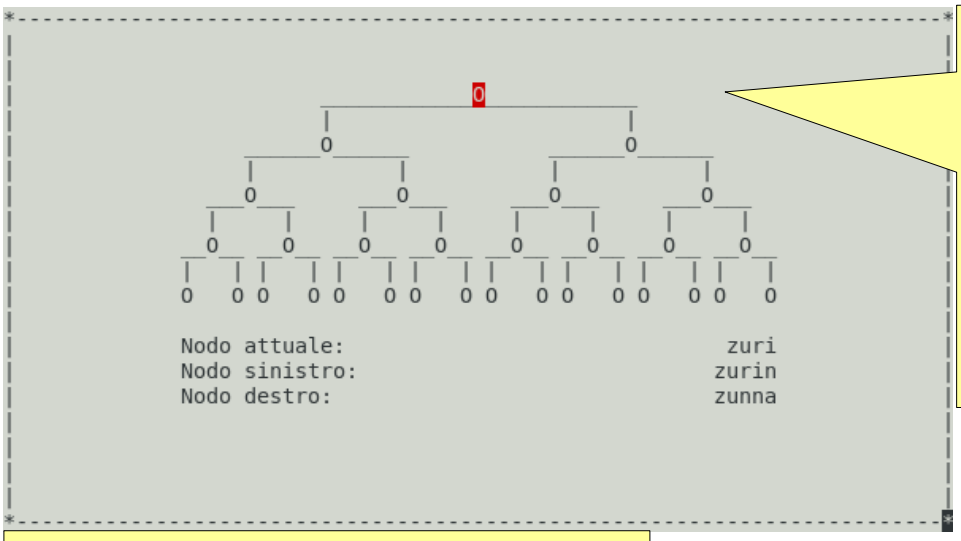
### *5.1.Manuale d'uso*

Segue un breve tutorial, nel quale illustriamo come generare un grafo, inserire un hub e ricercare percorsi. Vediamo così le principali interazioni con il programma:

**IMPORTANTE:**
in **tutte** le finestre di interazione utilizzare i seguenti controlli:
Tasti Direzionali : per muoversi nei menù;
Backspace : Indietro;
Enter: Esegui Operazione.

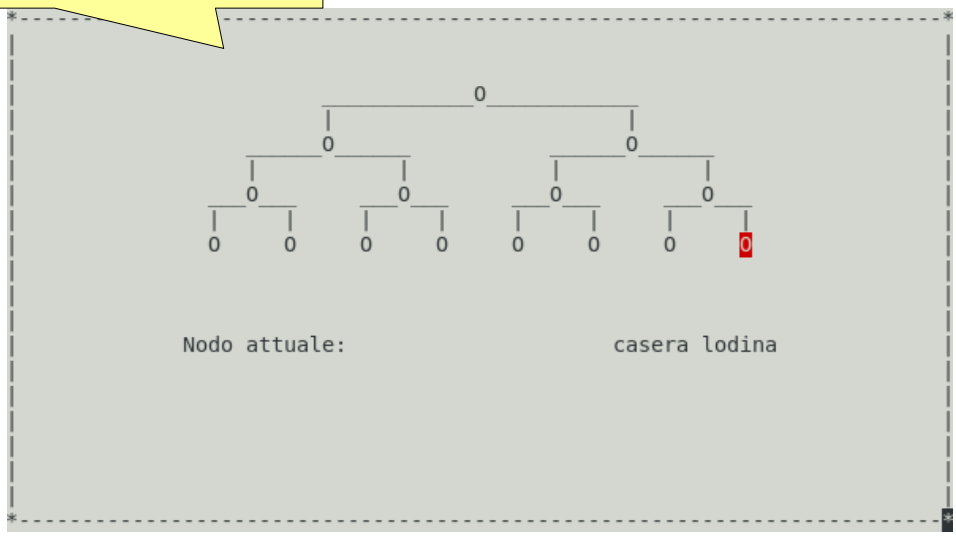


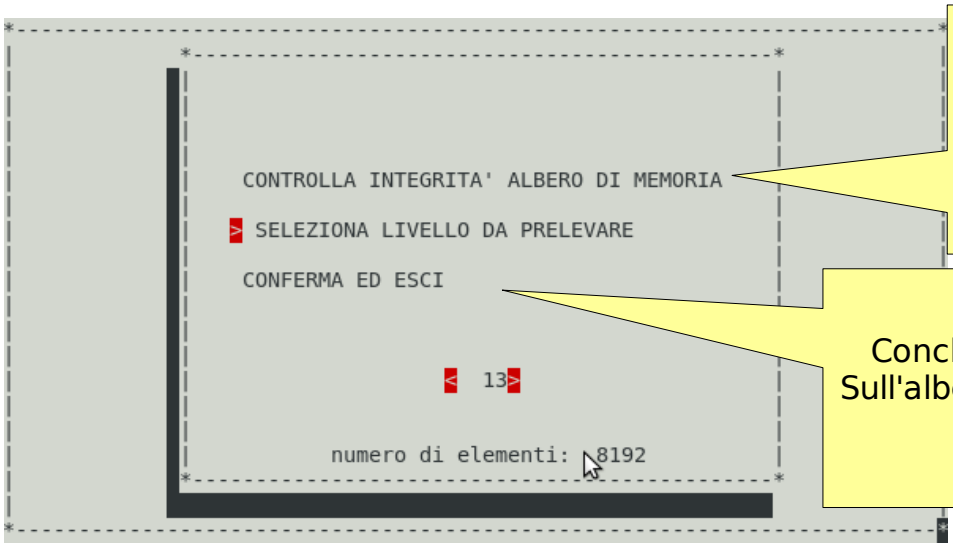






All'inizio il cursore
È posizionato a root
Usare i tasti
Direzionali per
Scorrere il “lotto”
Dell'albero

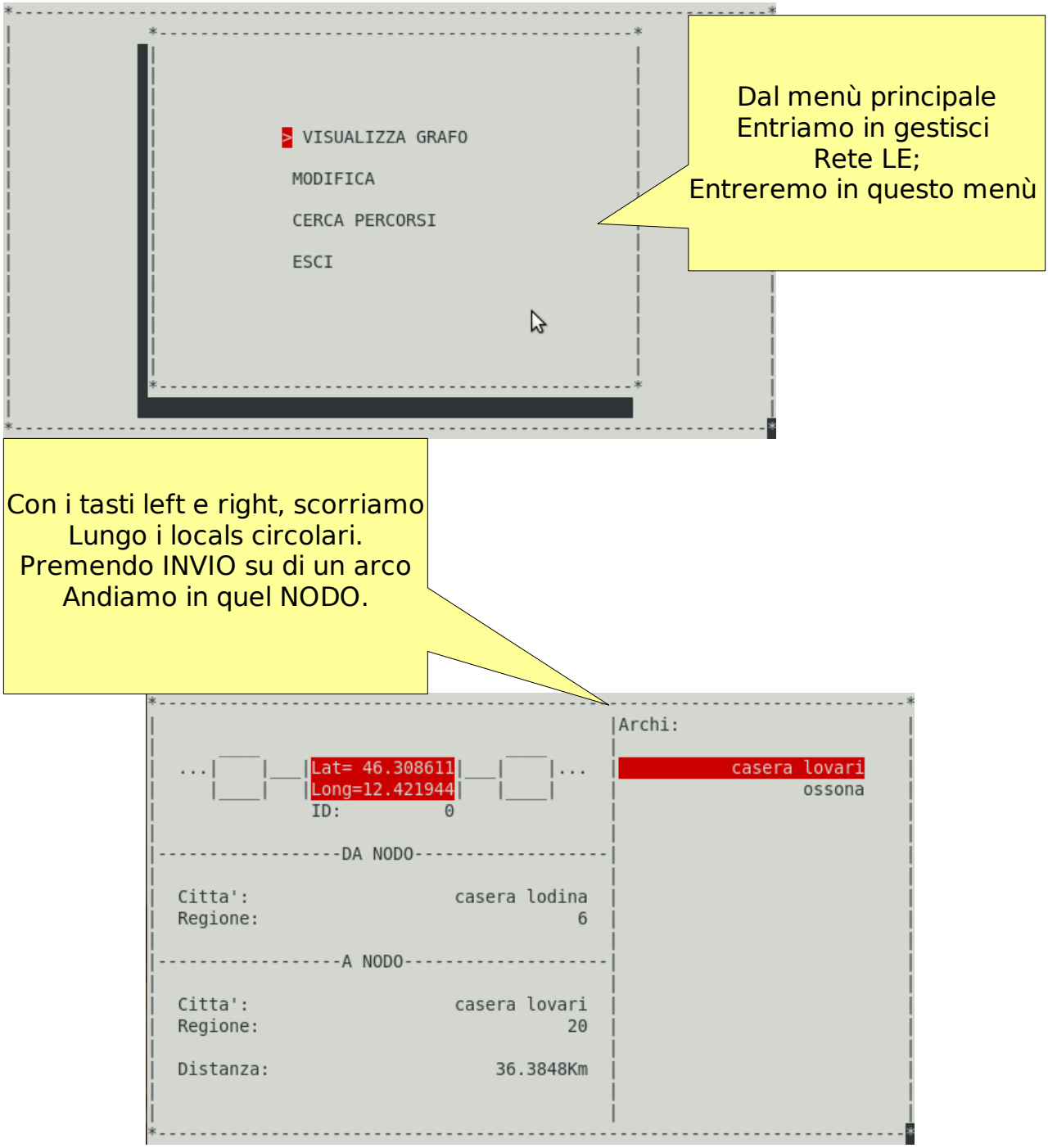
Arrivati nell'ultima posizione della
Riga, il cursore ricompare in alto, ma
Inizia la visualizzazione di un nuovo
Lotto dell'albero




Menu di Estrazione
Possiamo scegliere
Il livello da estrarre

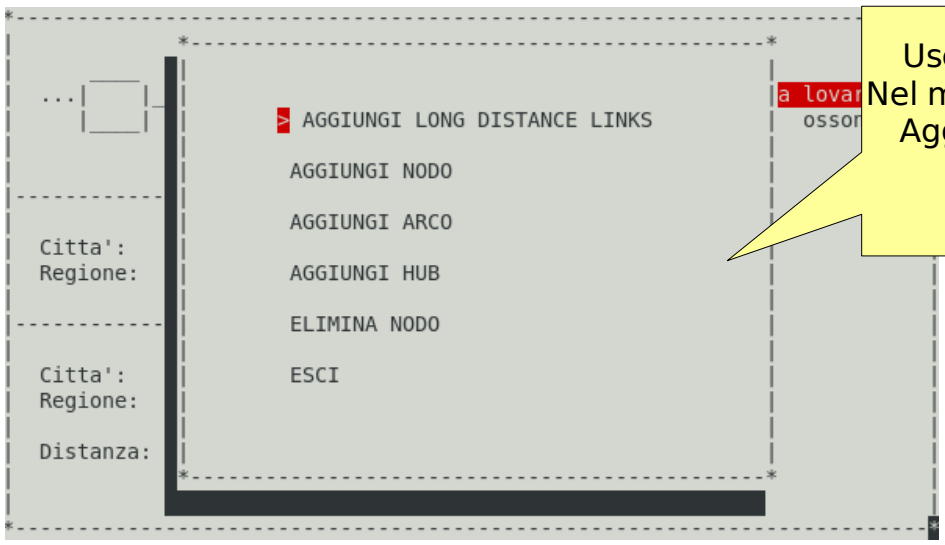
Concluse le operazioni
Sull'albero, confermiano e
usciamo.







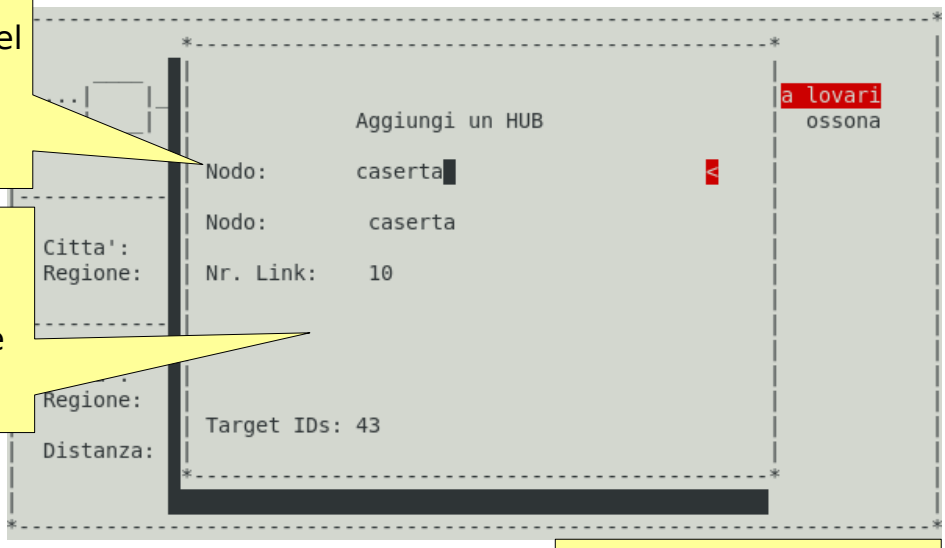
***Seggue alla pagina successiva...***




Usciamo e entriamo
Nel menù modifica grafo,
Aggiungiamo un HUB

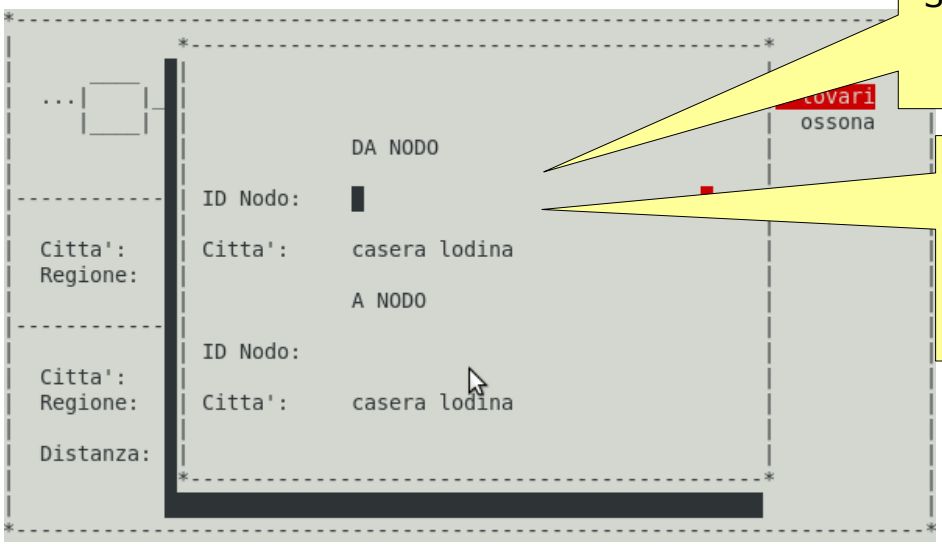
Scriviamo il nome del
Nodo e
premiamo INVIO

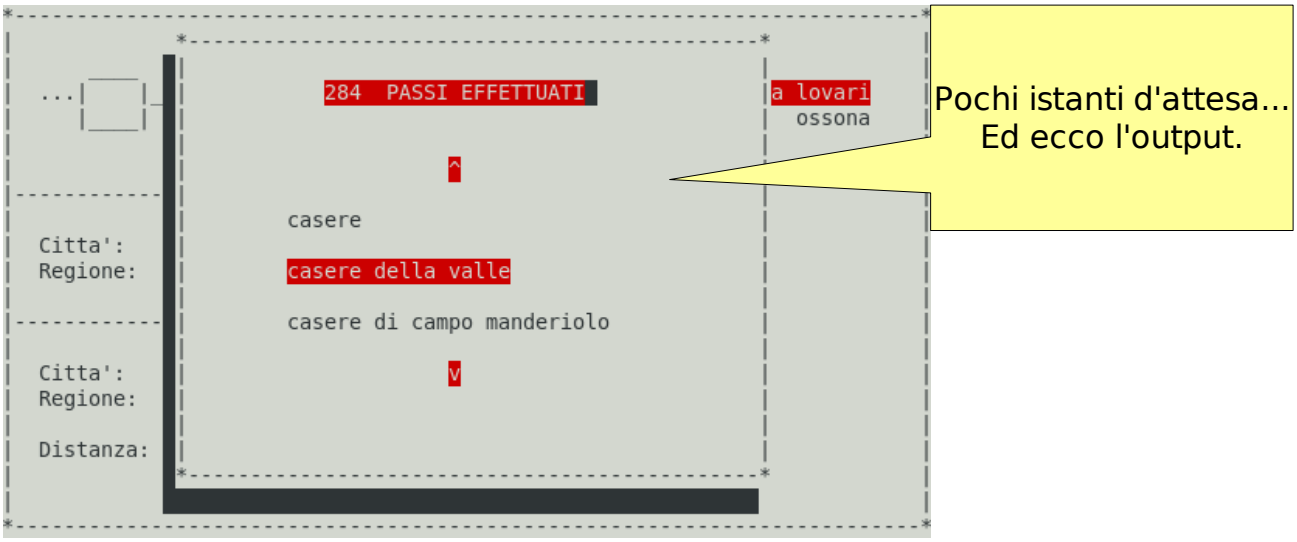
Con il tasto DOWN
Posizioniamoci
Sul numero di link e
scegliamo

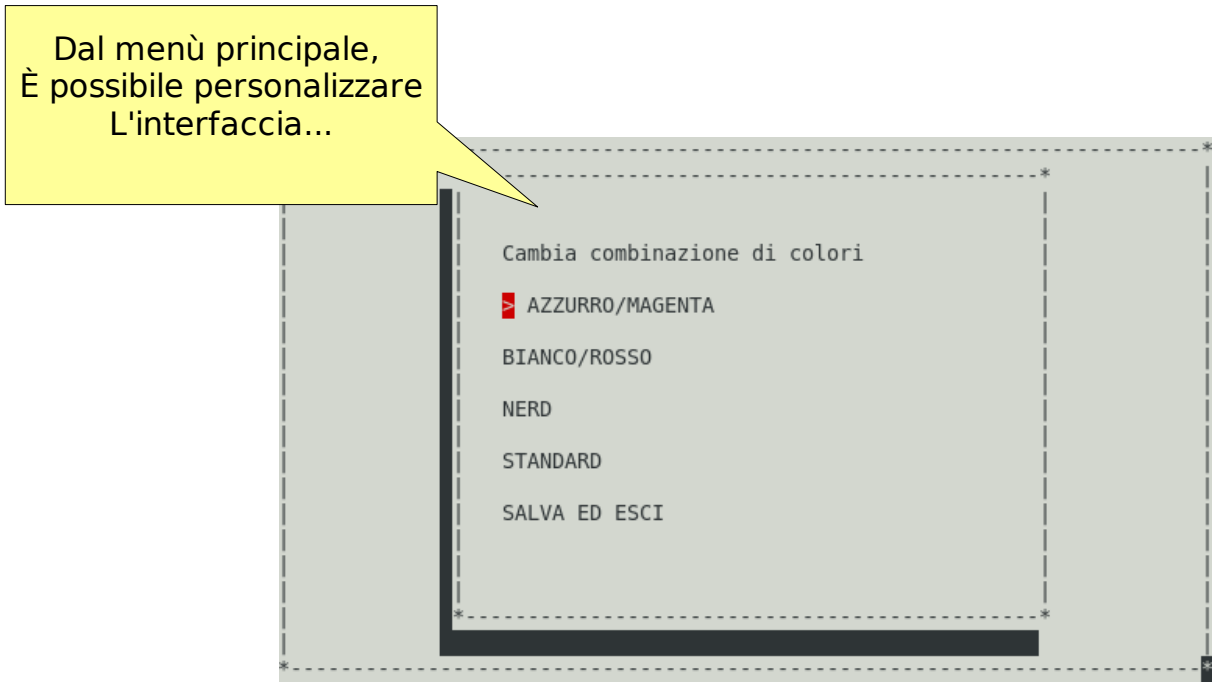



Dal menù gestione rete
Scegliamo di cercare
Dei percorsi

Scriviamo il nome dei
Nodi
premiamo INVIO






Speriamo di aver reso l'idea con questa breve dimostrazione.
Atre funzioni non esplicitamente dimostrate sono disponibili...buon divertimento(!?).

### *5.2.Codici Errore*
**[Non Presenti]**

## 6.Informazioni tecniche aggiuntive

### *6.0.Gestione Sviluppo*
Programma scritto testato e compilato sulle seguenti piattaforme:
GNU/Linux Ubuntu 8.10 Intrepid Ibex, con **gcc** alla versione **4.3.2**;
Intel®  Core Duo @1.60GHz.

GNU/Linux Ubuntu 8.04 Hardy Heron, con **gcc** alla versione **4.2.4**;
AMD® TurionX2 64 bit @1.60GHz.
Windows Vista, con **WxDevC++** (compilatore **gcc**, linker **MingW**).

### *6.1.Credits*
***Sviluppo, stesura e testing a cura di***
***Arcadio Ciro 5662675 - c.arcadio@studenti.unina.it ;***
***Cittadini Paolo 5662662 - p.cittadini@studenti.unina.it ;***

***ArCiSoft***
***info.okcomputer@gmail.com***