Progetto Diehard

Galasi Massimo - Matricola 747112

1 Analisi del problema

Il problema diehard è suddivisibile in più sottoproblemi :

- problema della generazione del grafo
- problema della visita del grafo
- problema della ricerca del cammino minimo

2 Strutture dati e funzioni

2.1 I contenitori

Per risolvere il problema della gestione dei contenitori e per mantenere in memoria tutte le possibili combinazioni elementari di svuotamento, riempimento e travaso dei contenitori si è pensato a una struttura di tipo *grafo orientato con liste di adiacenza*. La struttura mantiene in memoria il numero di vertici *nv*, il numero di archi *ne*, il numero di contenitori *nc*, l'array *capacita* contenente le capacità massime dei contenitori, l'array *attuale* contenente i livelli dei contenitori allo stato attuale, *sv_attivo* per indicare la possibilità di poter svuotare un contenitore, il parametro *pesato* per indicare se sono stati calcolati i pesi di ogni arco e infine l'array di puntatori a liste **nodi* per memorizzare i nodi.

```
typedef struct graph {

        int nv;
        int ne;
        int nc;
        int *capacita;
        int *attuale;
        int sv_attivo;
        int pesato;
        struct node **nodi;          /*array di puntatori a liste*/

} graph;
```

Da ogni vertice del grafo partono uno o più archi. Questi archi sono rappresentati attraverso una struttura *nodo* la quale contiene la configurazione raggiunta con una sola mossa. Notare che non si è fatta in realtà alcuna distinzione tra vertici del grafo e archi. Entrambi sono rappresentati con la stessa struttura *nodo*. La differenza sostanziale è che i vertici occupano sempre la prima posizione

delle liste di adiacenza e gli archi seguono sempre il vertice dal quale partono. Lo spazio richiesto è quindi O(|E|+|V|) e per verificare se i vertici *u* e *v* sono adiacenti il tempo richiesto è O(|V|).

```
typedef struct node {

        int v;
        char azione;
        int pericolosa;
        int costo;
        int colore;
        int *conf;
        struct node *next;          /* puntatore al nodo adiacente  */

} node;
```

In ogni nodo è contenuta la chiave univoca *v* e la configurazione raggiunta dei livelli *conf*. La chiave univoca è stata introdotta per facilitare le operazioni di ricerca ma si poteva considerare come chiave la configurazione stessa dei contenitori.
Altri dati satellite sono il tipo carattere *azione* $\in$ *{ 's' ,' r' , 't ' }* che memorizza il tipo di azione che è stata compiuta per ottenere la configurazione del nodo raggiunto. In questo modo possiamo per esempio identificare quelle configurazioni ottenute tramite uno svuotamento.
Il tipo intero *pericolosa* indica se la configurazione è stata segnalata dall'utente come pericolosa e quindi impraticabile.
Il tipo intero *colore* $\in$ *{ 0 , 1 , 2}* per sapere quali nodi sono già stati visitati. Sono stati pensati tre diversi livelli di colorazione per ciascun nodo: inizialmente tutti i nodi hanno colore *BIANCO (valore 0)*, se un nodo viene visitato assume quindi colore *GRIGIO (valore 1)* ma se il nodo visitato è il nodo che si sta cercando allora viene colorato di *NERO (valore 2)*. Quest'ultima colorazione è superflua ma utile per il riconoscimento delle configurazioni "goal".
Infine il tipo intero *costo* per attribuire un peso proporzionale alla lunghezza del percorso effettuabile da un nodo sorgente al nodo considerato.

2.2 Le operazioni sui contenitori

1. (X,Y: X < 5) -> (5,Y)	Riempi contenitore da 5 litri
2. (X,Y: Y < 3) -> (X,3)	Riempi contenitore da 3 litri
2. (X,Y: X > 0) -> (0,Y)	Svuota contenitore da 5 litri
3. (X,Y: Y > 0) -> (X,0)	Svuota contenitore da 3 litri
5. (X,Y: X+Y >= 5 and Y > 0) -> (5,Y-(5-X))	Travasa litri dal contenitore da 3 in quello da 5
6. (X,Y: X+Y >= 3 and X > 0) -> (X-(3-Y),3))	Travasa litri dal contenitore da 5 in quello da 3
7. (X,Y: X+Y <= 5 and Y > 0) -> (X+Y,0)	Svuota contenitore da 3 travasando nel contenitore da 5
8. (X,Y: X+Y <= 3 and X > 0) -> (0,X+Y))	Svuota contenitore da 5 travasando nel contenitore da 3
9. (X,Y: X > 0) -> (X-D,Y)	Riempi contenitore da 3 travasando nel contenitore da 5
10. (X,Y: Y > 0) -> (X,Y-D)	Riempi contenitore da 5 travasando nel contenitore da 3

Ad ogni operazione di riempimento, svuotamento e travaso si guardano i valori contenuti agli indici dell' array di configurazione *attuale* appartenente al grafo e si opera su di esso. Se vogliamo riempire o travasare la quantità di litri di un contenitore i e questa quantità è uguale a quella del contenitore i di *capacita* massima allora non verrà effettuata alcuna operazione (si è supposto che versare acqua fuori dai contenitori induca lo scoppio della

bomba) ; altrimenti, in presenza di una qualsiasi altra quantità inferiore alla capacità massima, verrà effettuato il riempimento assegnando al contenitore i-esimo il suo valore massimo e il travaso versando la quantità di acqua consentita. Si noti che per il travaso si è scelto di decrementare ciclicamente finché possibile la quantità dei litri versati. Questa scelta implica un costo maggiore rispetto all'assegnare direttamente il valore della capacità massima facendo un riempimento. Per i casi di riempimento e di svuotamento il costo è lineare perché si scandisce l'array della configurazione attuale e si effettua un confronto di costo unitario con l'elemento allo stesso indice dell' array capacita.

> Esempio di operazione valida:

Dato un contenitore j non pieno, un operazione valida di travaso da un contenitore i su un contenitore j decrementa la quantità di acqua di i finché il contenitore j non si riempie oppure finché il contenitore i non si svuota.

> Esempio di operazione non valida:

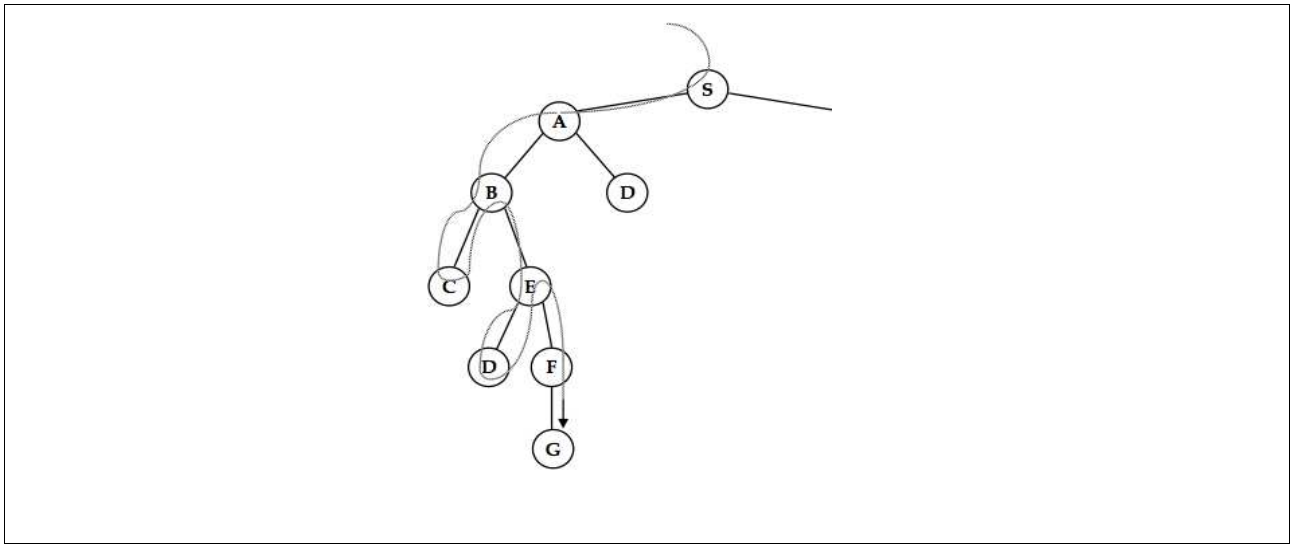
Un' operazione non valida è quella che vede uno svuotamento su un contenitore già vuoto oppure un riempimento su un contenitore già pieno.

Per visualizzare la configurazione attuale il tempo richiesto è sempre lineare, si stampano i valori dell' array *attuale.*

2.3 Visite in profondità

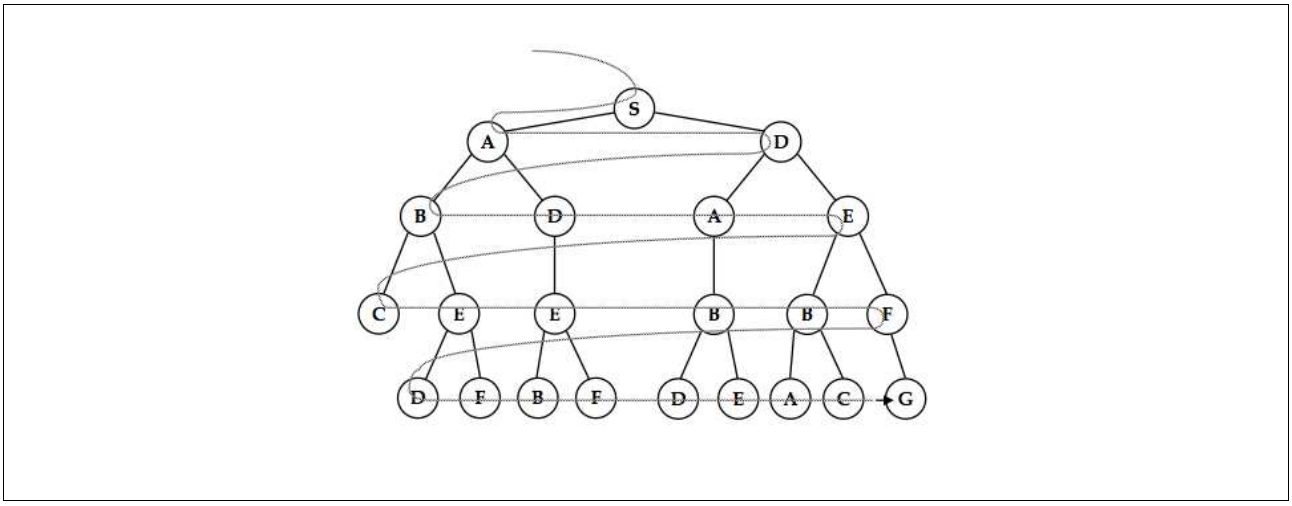
Per la funzione *configurazione* e la funzione *contenenti* si è scelto di utilizzare come strumento risolutivo la visita in profondità. Si parte da un vertice v e ricorsivamente si processano tutti i nodi adiacenti al vertice v. Bisogna evitare i cicli e lo si fa marcando (colorando) di volta in volta i nodi del grafo. Questa operazione ha un costo non indifferente dal momento che si è scelto di scorrere tutti i nodi del grafo e di marcare i nodi con la stessa chiave del nodo appena visitato con il colore grigio.
Una soluzione migliore poteva essere quella di memorizzare di volta in volta in un array ausiliario le chiavi dei nodi visitati e di effettuare per ogni nodo incontrato un confronto con le chiavi memorizzate in esso. *“Se la chiave del nodo che si sta visitando è uguale a una delle chiavi presenti in questo array, allora salta il nodo perché è già stato visitato”*.
Un'altra struttura ausiliaria ma sempre di tipo array di interi è stata utilizzata per contenere le chiavi dei nodi soluzione le cui configurazioni dovranno essere successivamente stampate.
L'algoritmo di visita in profondità è stato opportunamente modificato in entrambe le funzioni a seconda delle specifiche. *Configurazioni* in particolare scava in profondità fin quando il numero di mosse effettuate non è uguale a zero. Ad ogni nuova ricorsione il valore della variabile *mosse* viene decrementato e la dimensione dell'array contenente le chiavi trovate viene aggiornato. La funzione *contenenti* invece richiama se stessa ricorsivamente fino al completamento dell'intera visita del grafo. Ogni nuova chiamata ricorsiva passa il valore del livello che è stato fornito in input e la dimensione dell'array delle chiavi trovate aggiornato.

Il tempo è proporzionale a |V| + |E| in un grafo G=(V,E), quindi ha limite superiore asintotico O(|V|+|E|) . Se il problema ammette più di una soluzione, l'algoritmo non è detto che trovi quella a profondità minima. In altri termini l'algoritmo non è ottimale. Nel caso migliore, il nodo goal si trova alla estrema sinistra dell'albero generato. Quindi il numero di nodi da esaminare è: d+1 (per d = numero del livello).



2.4 Grafo pesato e calcolo del cammino minimo

Dati due nodi x e y, il problema del cammino minimo consiste nel fornire un cammino da x a y di peso minimo. Per la funzione *mosse* (con un solo parametro) si cerca il cammino lungo il grafo che a partire dalla configurazione attuale giunga a una configurazione contenente il valore k come livello di almeno un contenitore. Per risolvere questo problema è stato utilizzato l'algoritmo di Dijkstra. L'algoritmo di Dijkstra risolve il problema dei cammini minimi con sorgente singola su un grafo orientato e pesato G=(V,E) nel caso in cui tutti i pesi degli archi siano non negativi.
Ai cammini (orientati) nel grafo (cioè sequenze di archi consecutivi) verrà a questo punto assegnato un peso, dato dalla somma dei pesi degli archi che lo compongono.
Per fare questo ci si è serviti dell'algoritmo di visita in ampiezza che, procedendo per livelli, assegna ad ogni nodo il peso corrispondente al livello raggiunto. L'algoritmo *Breadth-first-search* fa uso di una coda per gestire l'insieme di vertici già esplorati, ogni vertice sarà inserito nella coda al più una volta. Le operazioni di inserimento e di eliminazione dalla coda richiedono tempo O(1) e il tempo dedicato per l'intera gestione della coda è O(V). Poiché il tempo speso per la scansione delle liste è pari a O(E), in totale si ha un tempo di O(V+E). La funzione *grafo_pesato* prende come parametro il grafo non pesato e ritorna il grafo con tutti i pesi assegnati per ogni nodo. Dopo aver assegnato peso 1 alla sorgente (il nodo che ha come configurazione dei livelli dei contenitori la configurazione attuale), *grafo_pesato* trasferisce il grosso del lavoro alla funzione *assegna_peso* la quale prosegue il percorso di visita fin quando tutti i nodi sono stati visitati.

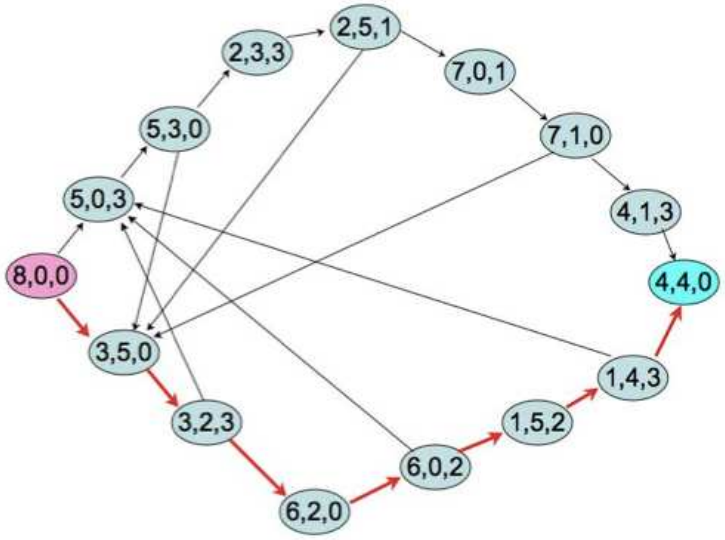


L'array di interi *livelli*, passato come parametro alla funzione *esiste_ragg_mosse_critica*, contiene i valori dei livelli di partenza e di destinazione. Se l'utente ha inserito solo il livello di destinazione, l'array *livelli* contiene questo valore e il valore -1 (usato come sentinella). L'algoritmo cerca quindi tutte le destinazioni possibili che hanno come livello di uno dei contenitori quello inserito. Se invece l'utente ha inserito sia il livello del nodo di partenza e sia il livello del nodo destinazione (il caso di *mosse* con due parametri) l'algoritmo calcola tutte le sorgenti e le destinazioni possibili. Poiché l'algoritmo di Dijkstra calcola il cammino minimo da una sorgente a una destinazione, la procedura di calcolo del cammino viene iterata un numero di volte pari al numero delle sorgenti e delle destinazioni, variando di volta in volta il valore della lunghezza minima trovata.
Due cicli for scorrono uno dentro l'altro, il primo ciclo, quello più esterno scorre per tutte le sorgenti trovate mentre il secondo scorre per tutte le destinazioni trovate. Ogni cammino così generato viene memorizzato in un array di interi *mst (arborescenza del cammino minimo)* e si effettuano successivamente dei controlli utilizzando la funzione *cammino_valido.* Quest'ultima, verifica se il cammino che è stato ottenuto contiene delle mosse non valide come quella di svuotamento di un contenitore nel caso sia stata disattivata la possibilità di svuotamento. Una volta ottenuto un cammino valido con percorso più breve lo si riscrive su un nuovo array definitivo *MST* . Se il percorso del cammino calcolato non è minore del percorso del cammino minimo calcolato fino a quel momento, non viene fatta alcuna operazione di controllo sul cammino e si passa alla prossima iterazione. A fine ciclo la variabile *pesomin* conterrà la lunghezza del percorso del cammino più breve, *MST* conterrà le chiavi dei nodi che compongono il cammino minimo trovato e nel caso *mosse* in particolare verrà stampata la configurazione di ogni nodo memorizzato nell'array *MST*.
La funzione *verifica_sorgente_destinazione* si occupa di calcolare le sorgenti e le destinazioni per le funzioni *mosse, esiste*, *raggiungibile* e *critica* e ritorna gli array di interi *sorg* e *dest* contenenti le chiavi delle sorgenti e delle destinazioni trovate. La complessità dell'algoritmo di Dijkstra è O(|E|log|V|).

Sia per le funzioni citate in questo paragrafo e le funzioni citate nel paragrafo precedente si è tenuto conto delle configurazioni considerate *pericolose* e quindi impraticabili per le visite in ampiezza, per le visite in profondità e per l'algoritmo di Dijkstra.

the shortest path from node (8,0,0) to (4,4,0).

The shortest path is shown below. The minimal number of transfers is 7.



2.5 Due algoritmi decisionali

Le funzioni *esiste* e *raggiungibile* si servono dell'algoritmo di Dijkstra per il calcolo del cammino minimo anche se non è necessario calcolare il cammino più corto. La decisione è stata presa per il reimpiego del codice utilizzato per la funzione *mosse*. Il tempo impiegato è quindi lo stesso di quello dell'algoritmo per la costruzione del cammino minimo e il tempo complessivo è minore del tempo complessivo impiegato dalla funzione *mosse* poiché, appena viene trovato un cammino minimo dalla sorgente a una qualsiasi delle possibili destinazioni, il ciclo di generazione di tutti i possibili cammini termina. Nel caso peggiore in cui non esiste nessun cammino che dalla sorgente permette di arrivare alla/e destinazione/i il costo sarà pari al costo massimo cioé il numero di iterazioni del ciclo for (per le destinazioni trovate) moltiplicato il costo di generazione del cammino minimo.

3 Commenti

Il programma è stato realizzato su sistema operativo Microsoft Windows Xp Professional 2002 ed è stato testato sia su Windows Xp che su Ubuntu Linux 8.04. Il programma prendendo in input i valori di esempio riportati sul testo del progetto, restituisce l'output corretto tranne che per l'azione critica che dovrebbe restituire la configurazione (3[3],3[5]). Se mi sarà consentito desidererei poter chiarire definitivamente questo problema irrisolto. Le strutture del grafo, delle liste e delle code sono state prese dal sito del professor Stefano Aguzzoli ed opportunamente modificate secondo le esigenze di sviluppo.