Cognome e nome: Fassio Davide
Matricola: s268519
Esame del 26/01/2021, Prova da 18 punti

Gerarchia del progetto

Il progetto è suddiviso in 3 moduli:

1. Tabella di simboli (tab_simb.h / tab_simb.c)
2. Lista (lista.h / lista.c)
3. Grafo (grafo.h / grafo.c)

I moduli 1,2 sono necessari per il modulo 3.
Le funzionalità del modulo grafo sono poi importate e utilizzate dal client main.c.

Strutture dati

Il PATH è stato implementato come quasi ADT nello stesso modulo del grafo.
Il campo vert (vertici) è un vettore di interi in cui ogni intero rappresenta l'indice di un nodo del grafo, il vettore è lungo len. L'intero prstes (preso tesoro) serve come flag per sapere se nel percorso è stato preso un tesoro (1) o meno (0). Il campo val (valore) contiene il valore del percorso, comprensivo del tesoro raccolto e della possibile penalità applicata per la fine del percorso. Infine, pf (punti ferita) rappresenta i punti ferita rimanenti alla fine del percorso.
L'allocazione della struttura è effettuata in GRAPHpathLoad e in GRAPHpathBest.
La liberazione della memoria è lasciata al client tramite la funzione PATH_free.

Il grafo è implementato come ADT di I categoria. È memorizzato tramite le liste delle adiacenze (liste linkate semplici) e essendo un grafo non orientato vale:

$$u \in ladj[v] \Leftrightarrow v \in ladj[u]$$

Il numero di vertici è salvato nel campo nnodi (numero di nodi). È utilizzata una tabella di simboli, ts, per la corrispondenza biunivoca $indice \Leftrightarrow nome$. La matrice vert (vertici) è utilizzata per salvare i dati relativi ai vertici, ha dimensione $[nnodi \times 3]$, ogni riga contiene una tupla:

(profondità, valore del tesoro, numero di monete d'oro)

L'allocazione della struttura è effettuata in GRAPHload.
La liberazione della memoria è lasciata al client tramite la funzione GRAPH_free.

Strategia risolutive

La funzione GRAPHload ha un funzionamento lineare: alloca le varie strutture, riempie la tabella di simboli (è la tabella di simboli a gestire eventuali ripetizioni), inserisce gli archi nelle liste delle adiacenze (sia l'arco che il suo inverso) e restituisce un puntatore al grafo così creato.

La funzione GRAPHpathLoad riceve in input un file nel formato:

```
Numero di vertici
Vert1 Vert2 … VertN
```

alloca il vettore di vertici e tramite la tabella di simboli del grafo salva gli indici dei vertici nel vettore, infine restituisce un puntatore al path così creato.

La funzione GRAPHpathCheck riceve in input il grafo, il path, il numero di mosse massimo e i punti ferita massimi. Inizia col controllare che il path inizi entrando nel labirinto e che sia più corto del numero massimi di mosse. Per ogni vertice toccato controlla se esiste l'arco di collegamento e se l'arco contiene una trappola la applica ai punti ferita, salva anche l'oro raccolto e il tesoro col valore maggiore. Finito il ciclo controlla se le condizioni di terminazione sono ammissibili, cioè se ho ancora delle mosse possibili e dei punti ferita spendibili allora l'unico modo per terminare è tramite l'uscita.

Appurato quindi che il percorso è ammissibile salva i valori calcolati nel PATH e applica le penalità in caso di terminazione in livelli con profondità diversa da zero.

La funzione GRAPHpathBest funge da wrapper per la funzione ricorsiva GRAPHbestPathR.
Nel wrapper si allocano due PATH, uno per la soluzione ottimale e uno per i calcoli, e un vettore (oro) per determinare in quali stanze le monete d'oro sono già state raccolte, viene poi chiamata la funzione ricorsiva, vengono liberate le strutture dati ausiliarie e infine viene restituito il percorso ottimale.
L'idea alla base della funzione ricorsiva è molto semplice:

- Se ho raggiunto una condizione di terminazione (finite le mosse possibili, esauriti i punti ferita oppure tornato all'uscita) aggiungo il tesoro dal valore massimo e applico le penalità. Se la soluzione corrente è migliore della soluzione ottima allora salvo la soluzione corrente in quella ottima.
- Se no percorro la lista delle adiacenze dell'ultimo vertice, inserisco il vertice, modifico i valori della soluzione corrente, ricorro e infine applico il backtrack.

Quando ho esaurito lo spazio delle possibilità avrò trovato la soluzione ottima.
Non viene effettuato nessun tipo di pruing.

Elenco commentato delle modifiche effettuate

Riga 17 del file grafo.h: modificato prsTes in prstes per uniformarmi allo stile usato nelle altre parti del programma.
Tutte le seguenti modifiche sono state effettuate nel file grafo.c:

- Riga 4: modificato il tipo delle liste di adiacenza da link a Lista visto che il modulo Lista.h è pensato per essere usato con il tipo Lista.
- Riga 6: cambiato il node della matrice da gain a vert per maggiore comprensibilità.
- Righe 95/98: aggiunto il conteggio delle penalità alla funzione GRAPHpathCheck.
- Riga 113: aggiunta la variabile tmpval che nelle righe 115/133 prende il posto di tmp->val in quanto modificare tmp->val andava a rompere il backtrack.
- Righe 117/123: adesso il controllo per il tesoro viene fatto prima dell'applicazione delle penalità così viene penalizzato anche il valore portato dal tesoro.
- Righe 126 e 128: aggiunto il .0 per forzare un cast a float e avere una moltiplicazione corretta.
- Riga 153: adesso inserisco il vertice in i e non più in i+1 per comodità e leggibilità.
- Riga 158: aggiunta la modifica al vettore oro in quanto prima non veniva mai modificato il suo stato.
- Riga 162: modificato l'indice da i a u, errore di battitura.
- Riga 172: nel backtrack sottraggo correttamente il valore aggiunto.
- Riga 192: ora la lunghezza del path corrente parte correttamente da 1.
- Riga 193: ora il valore del path corrente parte correttamente dal valore della stanza d'ingresso.
- Riga 198: viene raccolto l'oro nella stanza d'ingresso.
- Riga 200: modificato i in 1, errore di battitura.

Le modifiche effettuate come si può notare non apportano un cambiamento al paradigma scelto durante la prova ma ne correggono alcuni dettagli sfuggiti per la fretta.