Esercitazione di laboratorio n. 10

(Caricamento sul portale entro le 23.59 del 24/01/2018 di 3 esercizi su 4.
È caldamente consigliato di svolgerli tutti)

Esercizio n. 1: Tower defense

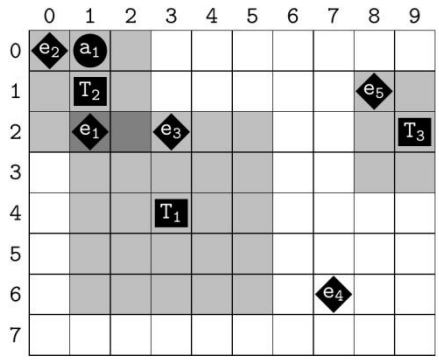
Con il termine *tower defense* si intende un sotto-genere dei giochi di strategia in cui il giocatore deve posizionare una serie di risorse (spesso delle torri) su una mappa, per raggiungere un determinato obiettivo.
Nel caso di questo esercizio, si consideri una situazione in cui la mappa di gioco è caratterizzata dalla presenza di alcuni target da difendere (alleati) o distruggere (nemici). Il giocatore deve decidere quali risorse acquistare dato un budget iniziale `B` e successivamente posizionare le risorse comprate su una mappa `R*C` per proteggere i suoi alleati e distruggere i suoi nemici.

Si considerino le seguenti indicazioni:

- ogni risorsa `T` è caratterizzata da un raggio di azione `r=[1..5]`, un parametro di attacco `a=[1..3]`, un parametro di difesa `d=[1..3]` e un costo di acquisto `c=[1..25]`
- ogni risorsa piazzata sulla mappa applica i propri parametri di attacco/difesa su tutta l'area coperta dal suo raggio di azione
- per semplicità si assuma che l'area di influenza di una risorsa sia un'area quadrata, con al centro la risorsa stessa, di dimensione `(2*r+1)*(2*r+1)`
- se più risorse coprono una medesima cella della mappa, il loro effetto è additivo
- una risorsa non può occupare la posizione in cui è situata un'altra risorsa o un target
- una risorsa è acquistabile in molteplici copie se il budget lo permette
- ogni target alleato a_i o nemico e_j è caratterizzato dalla sua posizione `<r,c>` sulla mappa e da un valore di riferimento `v=[1..10]`
- un target da difendere (distruggere) viene difeso (distrutto) se l'effetto cumulativo delle risorse che lo influenzano è maggiore o uguale al valore di riferimento del target stesso
- target alleati (nemici) non vengono attaccati (difesi) dalle risorse posizionate. In altre parole non c'è effetto di "fuoco amico", o il suo opposto.

Si scriva un programma in C che dato un budget, un elenco di risorse acquistabili e una configurazione per i target sulla mappa, scelga quali e quante risorse acquistare e individui, se esiste, un posizionamento delle risorse scelte tale per cui tutti i target alleati siano difesi e tutti i target nemici siano distrutti.

A titolo di esempio, si consideri la seguente configurazione parziale:



Nell'immagine si hanno:

- 3 risorse
  - T_1 a raggio 2
  - T_2 a raggio 1
  - T_3 a raggio 1
- 1 target alleato a_1
- 5 target nemici $e_1..e_5$

T_1 e T_2 coprono una porzione comune della mappa, per cui i loro effetti sono sommati in quelle celle. Cumulativamente, la somma dei valori in attacco di T_1 e T_2 deve almeno eguagliare il valore di riferimento di e_1 per poterlo distruggere. T_2 essendo l'unica risorsa a coprire il target a_1 deve avere un parametro di difesa almeno uguale al valore di riferimento del target stesso. e_4 non è coperto da nessuna risorsa in attacco, per cui la configurazione automaticamente non soddisfa i vincoli imposti dal problema.

Suggerimenti:
si individuano i seguenti sottoproblemi:

- individuare le risorse (anche in più copie) da utilizzare compatibilmente con il budget
- piazzarle sulla mappa
- verificare se la collocazione soddisfa i vincoli (di attacco e di difesa).

Si suggeriscono 2 strade:

1. separare l'individuazione delle risorse dal loro collocamento sulla scacchiera. Come primo passo si identifichi il modello del Calcolo Combinatorio che permette di elencare le risorse comprate compatibilmente con il budget. Successivamente si individui il modello del Calcolo Combinatorio che permette di piazzare ciascuna delle configurazioni individuate al punto precedente sulla mappa. Infine si scriva la funzione di validazione della soluzione così trovata.
2. selezionare (compatibilmente con il budget) e collocare contestualmente le risorse sulla mappa. Si noti che, estendendo l'insieme delle risorse di cardinalità *n* con la risorsa nulla, il problema si riduce a calcolare le disposizioni con ripetizione delle *n+1* risorse estese prese a `R*C` a `R*C`.

In entranbi i casi la verifica o viene fatta nel caso terminale o viene (anche solo parzialmente) anticipata con il pruning. La definizione di una strategia di pruning è facoltativa.

Esercizio n. 2: Bike sharing

Si richiede lo sviluppo di un'applicazione in grado di gestire un sistema di bike sharing secondo le seguenti modalità:

- ogni utente ha una tessera caratterizzata da un identificatore univoco (`cardId`) costituito da un stringa alfanumerica di 10 caratteri
- ogni stazione di prelievo/deposito bici ha un identificatore univoco (`stationId`) costituito da una stringa alfanumerica di 10 caratteri
- ogni stazione ha un lettore di tessere per abilitare un utente al prelievo (inizio noleggio) o deposito (fine noleggio) di una bicicletta
- ogni stazione ha a disposizione un certo numero di parcheggi `numPosti` e di biciclette `numBici`. I posti rimangono fissi per tutta la durata del programma mentre il numero di biciclette è variabile sulla base di noleggi e depositi
- è possibile avviare un noleggio solo se ci sono veicoli disponibili ed è possibile terminare un noleggio solo se ci sono parcheggi disponibili.

Il sistema riceve da tastiera le letture delle stazioni con formato:

```
stationId cardId time
```

Il tempo viene introdotto quale valore intero rappresentante il numero di minuti trascorsi a partire dalle ore `00:00` del giorno stesso. Per semplicità si assuma che non è possibile noleggiare una bici a cavallo tra due giorni (i noleggi attivi si chiudono prima della mezzanotte).

Alla lettura di una tessera il sistema deve:

- verificare se l'utente ha un noleggio già attivo
- se l'utente ha un noleggio attivo e la stazione ha parcheggi liberi, chiudere il noleggio corrente e ritornare la durata, in minuti, del noleggio stesso. In assenza di posti liberi, la chiusura del noleggio deve essere negata
- se l'utente non ha un noleggio attivo, avviare un nuovo noleggio se ci sono biciclette disponibili. In assenza di biciclette, l'avvio del noleggio deve essere negato.

L'elenco di stazioni è letto da file, il cui nome viene ricevuto dall'applicazione come parametro sulla linea di comando e ha il seguente formato:

- sulla prima riga appare il numero `S` di stazioni
- seguono `S` triple `stationId numBici numPosti`, dove `numBici` è un valore intero che rappresenta il numero di biciclette inizialmente disponibili in una data stazione e `numPosti` (maggiore o uguale a `numBici`) rappresenta il numero effettivo di parcheggi disponibili in una data stazione.

Il sistema, una volta inizializzato, deve:

- mantenere l'elenco di tutti gli utenti che abbiano mai fatto uso di una carta
- mantenere l'elenco di tutti le stazioni da cui ciascun utente abbia avviato un noleggio e per ciascuna stazione il numero di noleggi ivi iniziati
- fornire una funzione di lettura tessere del tipo

```
int leggiTessera(char *cardId, char *stationId, int time);
```

Si osservi che:

- essendo il numero di potenziali utenti relativamente elevato e variabile, il costo asintotico di tutte le operazioni effettuate deve essere al più logaritmico nel numero di utenti. A tale scopo, si imposti la soluzione appoggiandosi a un BST (ADT di I classe) di utenti, avente come criterio di ordinamento il loro `cardId`
- si consiglia poi, per ogni utente, di mantenere un vettore o una lista delle stazioni utilizzate (mantenendo, per ognuno, i dettagli richiesti precedentemente)
- essendo il numero di stazioni limitato e di numero noto, il costo asintotico di tutte le operazioni non ha alcun vincolo in relazione al numero di stazioni.

Esercizio n. 3: Rete di elaboratori

Un grafo non orientato e pesato rappresenta una rete di elaboratori appartenenti ciascuno ad una sottorete. Il peso associato ad ogni arco rappresenta il flusso di dati tra due elaboratori della stessa sottorete o di sottoreti diverse, come nell'esempio seguente (cfr. figura successiva).

Il grafo è contenuto in un file, il cui nome è passato come argomento sulla linea di comando.
Il file ha il seguente formato:

- sulla prima riga un unico intero `N` rappresenta il numero di vertici del grafo
- seguono `N` righe ciascuna delle quali contiene una coppia di stringhe alfanumeriche, di al massimo 30 caratteri, `<id_elab> <id_rete>`

- sulle righe successive, in numero indefinito, appaiono terne nella forma `<id_elab1> <id_elab2> <flusso>`

Si facciano anche le seguenti assunzioni:

- i nomi dei singoli nodi sono univoci all'interno del grafo
- non sono ammessi cappi
- tra due nodi c'è al massimo un arco (non è un multigrafo)
- le sotto-reti sono sotto-grafi non necessariamente connessi.

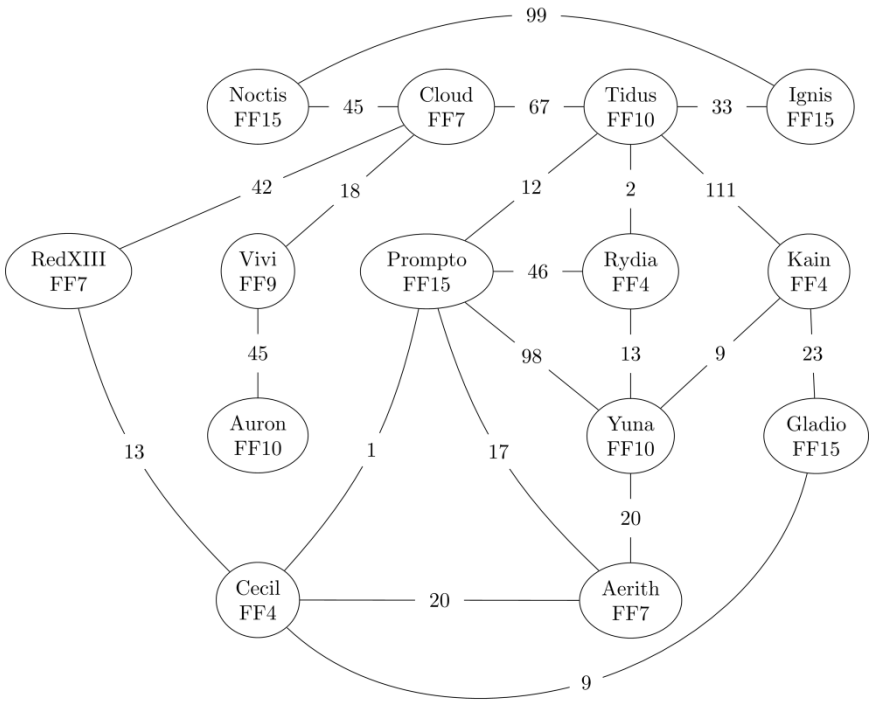
Si scriva un programma in C in grado di caricare in memoria il grafo, leggendone i contenuti da file e di potervi effettuare alcune semplici operazioni.
La rappresentazione della struttura dati in memoria deve essere fatta tenendo conto dei seguenti vincoli:

- il grafo sia implementato come ADT di I categoria, predisposto in modo tale da poter contenere sia la matrice sia le liste di adiacenza. Nella fase di caricamento dei dati da file si generino solamente le liste di adiacenza, su comando esplicito va generata anche la matrice di adiacenza
- il grafo sia realizzato identificando ogni nodo con un indice intero. Le informazioni relative agli elaboratori e alle sotto-reti siano memorizzate in un vettore di `struct`, tale da attribuire ad ogni nodo un indice intero (corrispondenza indice-dato) tra `0` e `|V|-1`
- gli indici siano generati automaticamente man mano che si acquisiscono i dati. A tale scopo si utilizzi una tabella si simboli (ADT di I classe) tale da fornire corrispondenze "*da nome a indice*". Internamente all'ADT le conversioni nome-indice siano gestite mediante una tabella di hash (realizzata con open addressing e linear probing).

Sul grafo, una volta acquisito da file, sia possibile:

- stampare il numero totale di vertici elencandoli esplicitamente per nome
- stampare il numero di archi incidenti su un nodo e l'elenco di vertici ad esso connessi
- generare la matrice di adiacenza, **SENZA** leggere nuovamente il file, a partire dalle liste di adiacenza
- determinare l'ammontare complessivo dei flussi *intra-* ed *inter*-rete. Si noti che per ottenere questi due valori è sufficiente percorrere iterativamente tutti gli archi del grafo.

In allegato al testo è presente il grafo d'esempio (`grafo.txt`) rappresentato a seguire:



Esercizio n. 4: Cammino nel labirinto

Un file di testo contiene una matrice di caratteri con il seguente formato:

- la prima riga del file specifica le dimensioni della matrice (numero di righe `nr` e numero di colonne `nc`)
- ciascuna delle `nr` righe successive contiene gli `nc` valori corrispondenti a una riga della matrice, separati da uno o più spazi
- le celle percorribili sono rappresentate dal carattere '`-`', i muri dal carattere '`X`', il singolo punto di ingresso dal carattere '`I`' e le una o più uscite del labirinto dal carattere '`U`'.

Si scriva un programma C che individui ricorsivamente la sequenza di passi minima, se esiste, per raggiungere una delle uscite disponibili (la più vicina) rispetto al punto di partenza dato.

Suggerimento: si osservi come la matrice rappresenti implicitamente un grafo, sul quale possono essere applicati gli algoritmi noti dalla letteratura.