Politecnico di Torino

03AAXOA

Algoritmi e Strutture Dati

Temi d'esame

Indice

1 22/06/2020

1.1 Traccia da 12 punti

1.1.1 3pt

Siano date 2 matrici di interi $\texttt{M}_1$ e $\texttt{M}_2$, di dimensioni rispettivamente $\texttt{r}_1 \times \texttt{c}_2$ e $\texttt{r}_1 \times \texttt{c}_2$. Si scriva una funzione `submat` che identifichi la più grande sottomatrice quadrata comune di $\texttt{M}_1$ e $\texttt{M}_2$ e le sue dimensioni. Tale sottomatrice deve essere memorizzata in una terza matrice `res`, allocata della dimensione opportuna. Il prototipo della funzione `submat` sia:

```
int **subMat(int **M1,int r1,int c1,int **M2,int r2,int c2,int *dim);
```

Esempio

$$M_1 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 1 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 4 & 4 \end{pmatrix} \; M_2 = \begin{pmatrix} 1 & 9 & 3 & 5 \\ 9 & 2 & 3 & 5 \\ 1 & 2 & 4 & 5 \end{pmatrix} \; res = \begin{pmatrix} 2 & 3 \\ 2 & 4 \end{pmatrix}$$

1.1.2 3pt

Siano dati 2 vettori di interi (positivi, negativi o nulli) $\texttt{v}_1$ e $\texttt{v}_2$ di lunghezza rispettivamente $\texttt{d}_1$ e $\texttt{d}_2$. Si scriva una funzione `prodCart` che generi una lista concatenata `L` contenente il risultato del prodotto cartesiano dei due vettori. La lista sia ordinata in fase di costruzione per prodotto crescente degli elementi di ogni coppia. Il prototipo della funzione `prodCart` sia:

```
list_t prodCart(int *v1, int d1, int *v2, int d2);
```

Si definisca la lista come ADT di I classe e il tipo nodo come quasi ADT. Si indichi esplicitamente in quale modulo/file appare la definizione dei tipi proposti. **È vietato l'uso di funzioni di libreria**.

Esempio

$v_1 = (1, 2, 3)$
$v_2 = (3, 4, 5, 6)$
$L = (1, 3), (1, 4), (1, 5), (2, 3), (1, 6), (2, 4), (3, 3), (2, 5), (3, 4), (2, 6), (3, 5), (3, 6)$

1.1.3 6pt

Sia dato un set di `N` oggetti, numerati da 1 a `N`. Ogni oggetto è caratterizzato da un suo nome e da una quaterna di interi positivi o nulli: `costo`, `valore`, `tipo`, `quantità`. Si scriva una funzione ricorsiva in grado di identificare un insieme di oggetti tale da rispettare i seguenti vincoli:

- Costo complessivo al più `C`, dove `C` è un intero passato come parametro
- Insieme composto solo da oggetti tipologie diverse
- Si termini l'esecuzione non appena trovata una soluzione valida
- Si giustifichi la scelta del modello combinatorio adottato
- Si giustifichi la scelta del/dei criteri di pruning adottati, o il motivo della loro assenza.

Il prototipo della funzione ricorsiva sia il seguente:

```
int solve_r(item_t *v, int n, int *sol, int pos, int tot_c, int tot_v,
    ...);
```

1.2 Traccia da 18 punti

1.2.1 Breve introduzione e definizioni

Una matrice rettangolare di dimensione `R` $\times$ `C` rappresenta uno schema (a griglia) per un gioco in stile "parole crociate" (o "cruciverba"), che comprende:

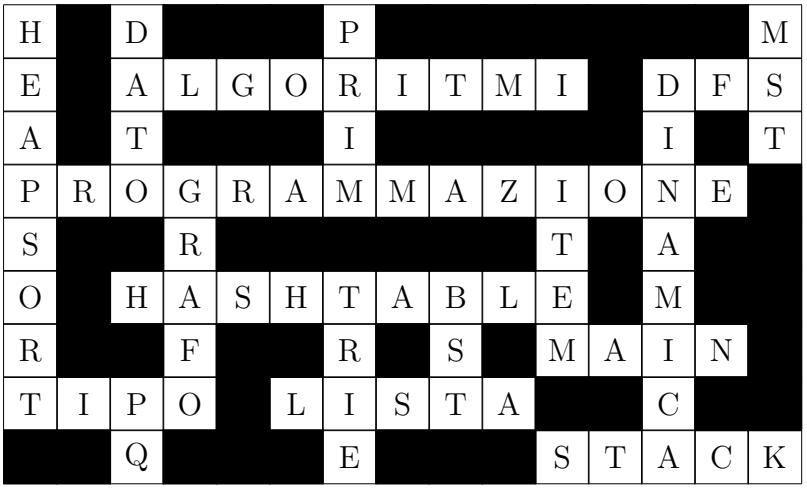
- Caselle nere, in cui non è possibile scrivere nulla
- Caselle bianche, in cui vanno scritti caratteri alfabetici (si supponga, per semplicità, di usare solamente lettere maiuscole).

Una volta completata, la griglia deve essere tale per cui ogni sequenza di almeno due caselle bianche (sia in orizzontale che in verticale) deve contenere parole "valide", non prolungabili (a sinistra/destra le orizzontali, sopra/sotto le verticali). Il gioco consiste nel collocare nella griglia (in orizzontale e in verticale) parole, scelte da un elenco dato, in modo tale da non lasciare caselle bianche vuote. Si noti che:

- Ogni parola orizzontale dovrà iniziare in una casella bianca priva di una casella bianca adiacente a sinistra, e terminare in una casella bianca priva di casella bianca adiacente a destra. Ragionamento analogo/duale va fatto per le parole in verticale.

- Gli incroci tra parole orizzontali e verticali vanno rispettati, cioè una casella bianca può appartenere sia a una parola verticale che orizzontale.

Esempio



Si vogliono risolvere in C i seguenti problemi:

- Definire una opportuna struttura dati in grado di rappresentare lo schema di gioco, utilizzabile per risolvere i problemi che seguono
- Verificare che uno schema già completato sia compatibile con un dato elenco di parole.
- Dato uno schema libero e un elenco di parole, trovare (se esiste) un possibile completamento dello schema mediante la selezione di un sottoinsieme delle parole.

1.2.2 Struttura dati e formato file

Tra i vari formati possibili per rappresentare uno schema, si è scelto quello di elencare in modo esplicito collocazione e lunghezza delle parole orizzontali e verticali. Date queste, le caselle nere, se necessario, possono essere determinate in modo univoco. Lo schema è rappresentato in un file che contiene:

- Prima riga, due interi `R` e `C` che rappresentano le dimensioni ed un intero `N` che rappresenta il numero di parole

- La collocazione delle parole orizzontali e verticali nello schema, una per riga, nel formato `<lunghezza> <r> <c> <direzione>`, dove `<lunghezza>` è la lunghezza della parola, `<r>` e `<c>` sono gli indici della casella di inizio e `<direzione>` è uno tra i caratteri `'O'` (per orizzontale) oppure `'V'` (per verticale).

Esempio

```
15 9 18 // Schema 15x9 con 18 punti di inizio per le parole
8 0 0 V // Parola verticale iniziante in posizione [0,0] di lunghezza 8 caratteri
4 0 2 V
4 0 6 V
...
5 7 5 O // Parola orizzontale iniziante in posizione [7,5] di lunghezza 5 caratteri
2 7 2 V
5 8 10 O
```

Per completare lo schema, è fornito un insieme di parole. Le parole sono riportate in un secondo file con il seguente formato:

- Sulla prima riga appare il numero `P` di parole
- Seguono `P` righe riportanti una parola ognuna.

Esempio

```
50
ALGORITMI
PROGRAMMAZIONE
DIJKSTRA
BST
UNIONFIND
...
```

Le parole hanno lunghezza massima 20 caratteri e non si ripetono nel file. Si definiscano due strutture dati:

- una (con wrapper `struct schema`) in grado si gestire lo schema, cioè le collocazioni (per le parole) nella griglia, sia orizzontali che verticali, ognuna caratterizzata da casella di inizio e da lunghezza.
- una (con wrapper `struct parole`) adatta a immagazzinare gli elenchi delle parole ammesse, organizzate per lunghezza, in modo tale da poter accedere con costo $O(1)$ all'elenco delle parole di una data lunghezza

Si scrivano le funzioni, per l'acquisizione di tali strutture dati dai file.

1.2.3 Funzione di verifica

Si scriva una funzione che, ricevuti come parametri uno schema, un elenco di parole e una matrice di caratteri (di dimensione compatibile con lo schema) contenente uno schema di cruciverba completato (il contenuto delle caselle nere è trascurabile, in quanto vanno solo controllate le parole), verifichi se le parole orizzontali e verticali presenti nello schema sono compatibili con l'elenco delle parole. Il prototipo della funzione sia:

```
int verificaSchema(schema *s, parole *p, char **m);
```

1.2.4 Problema di ricerca

Scopo del gioco è di posizionare opportunamente un sottoinsieme delle parole nella griglia completandola. Le parole non possono comparire più di una volta nella griglia. È sufficiente individuare la prima soluzione valida trovata, ammesso che esista. Individuata la prima soluzione, la funzione termina.

- Si dica a quale schema di funzione ricorsiva si fa riferimento e perché
- Si definisca la struttura dati usata per rappresentare la soluzione struct soluzione
- Si scriva la funzione wrapper di invocazione della funzione ricorsiva di ricerca

  ```
  void solve(schema *s, parole *p);
  ```

- Si scriva la funzione ricorsiva di ricerca che trova (se esiste) un elenco di parole in grado di completare lo schema

  ```
  void solve_r(schema *s, parole *p, soluzione *sol, int pos, ..., int
      *trovato);
  ```

- Si scriva una funzione che data una soluzione, verifichi se questa rispetta i vincoli (ad esempio, una parola verticale e una orizzontale devono avere lo stesso carattere al loro incrocio) oppure no. Una soluzione è data dalle parole selezionate per completare lo schema; si può trattare di soluzione parziale o completa, a seconda che si richiami la funzione a scopo di pruning oppure in un caso terminale della ricorsione. Completare opportunamente il seguente prototipo e l'implementazione della funzione sulla base della scelta di utilizzo effettuata.

  ```
  int checkSol(schema *s, soluzione *sol, ...);
  ```

2 03/09/2020

2.1 Traccia da 12 punti

2.1.1 2pt

La funzione `mySum` riceve in input due matrici quadrate di interi `A` (di dimensione `nA` $\times$ `nA`) e `B` (di dimensione `nB` $\times$ `nB`). Considerando per ciascuna matrice le righe e le colonne, sono possibili le seguenti coppie dove il primo elemento della coppia appartiene ad `A` e il secondo a `B`: (riga$_i$A, riga$_j$B), (riga$_i$A, col$_t$B), (col$_k$A, riga$_j$B), (col$_k$A, col$_t$B) dove $0 \leq i < nA, 0 \leq k < nA, 0 \leq j < nB, 0 \leq t < nB$. Si scriva un programma che identifica la coppia in cui la somma di tutti i valori del primo e del secondo elemento (siano essi righe o colonne) sia la più vicina possibile a zero. La scelta per ogni matrice sia espressa come stringa, nella forma `R<indice>` (`C<indice>`) per indicare un indice di riga (colonna). Gli indici partono da zero.

Esempio

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & -3 \\ 11 & 12 & -6 \end{pmatrix} B = \begin{pmatrix} 2 & 0 \\ 7 & 1 \end{pmatrix}$$

La coppia identificata dall'algoritmo è $(col_2A, riga_1B)$ per cui vale $(3 - 3 - 6) + (7 + 1) = 2$. L'output è quindi `C2 R1`. Si completi adeguatamente il codice della funzione, dato il prototipo:

```
coppia mySum(int **A, int nA, int **B, int nB)
```

e la definizione:

```
typedef struct coppia_ {char *scelta_A; char *scelta_B;} coppia;
```

Si riporti esplicitamente l'invocazione della funzione `mySum` fatta dal main chiamante.

2.1.2 4pt

Si definisca una struttura dati adatta a rappresentare una lista linkata circolare. In una lista circolare il puntatore al successore dell'elemento in coda punta all'elemento di testa della lista stessa. I nodi della lista siano caratterizzati da un valore intero e da un contatore di occorrenze. Si definisca la lista come ADT di I classe e il tipo nodo come quasi ADT. Indicare esplicitamente in quale modulo/file appare la definizione dei tipi proposti. **Non è ammesso l'uso di funzioni di libreria**. Si implementi la seguente funzione di inserimento nella lista definita in precedenza: se l'elemento è

già presente, si incrementa il contatore. Se non è presente va inserito nella posizione indicata dal secondo parametro. La testa ha posizione 0 per convenzione. **Non è ammesso l'uso di funzioni di libreria**.

```
void LISTinsert(list_t L, int posizione, ...)
```

2.1.3 6pt

Un locale ha a disposizione un insieme di `nS` sale. La capienza di ogni sala è registrata in un vettore `S`. Il locale riceve, per un certo giorno, prenotazioni da `nP` gruppi. La cardinalità di ciascun gruppo è registrata in un vettore `P`.

Esempio

4 sale di capacità, rispettivamente 4, 9, 11 e 5 persone:
$nS = 4$
$S = \{4, 9, 11, 5\}$

Prenotazione da parte di 9 gruppi con le rispettive cardinalità:
$nP = 9$
$P = \{2, 2, 3, 2, 6, 2, 4, 4, 3\}$

Si scriva un algoritmo ricorsivo in grado di determinare, se esiste, un'assegnazione delle prenotazioni ricevute nelle sale del locale, tale per cui il numero di sale occupate sia minimo. Si assuma non sia possibile spezzare una prenotazione su due/più sale. Il prototipo della funzione invocata sia nella forma:

```
void solve(int *S, int nS, int *P, int nP, ...);
```

Tale funzione può rappresentare un wrapper per l'effettiva funzione ricorsiva. Giustificare la scelta del modello combinatorio adottato. Giustificare la scelta del/dei criteri di pruning adottati, o il motivo della loro assenza.

2.2 Traccia da 18 punti

2.2.1 Breve introduzione e definizioni

Una matrice rettangolare di dimensione `R` $\times$ `V` rappresenta una griglia di gioco, le cui celle o sono vuote o contengono una cifra tra 1 e 9. La griglia di gioco è riportata su un file testuale (`griglia.txt`) con il seguente formato:

- sulla prima riga appare una coppia di interi `<R>` e `<C>` a rappresentare le dimensioni della griglia
- seguono `R` righe di `C` caratteri (tra 1 e 9 per le celle piene oppure 0 per quelle vuote).

Esempio

```
3 3
1 5 0
0 0 0
0 0 0
```

Lo scopo del gioco è completare la griglia riempiendo le celle vuote con valori tra 1 e 9 tali da creare regioni contigue di dimensione pari alle cifre in esse contenute. Si ricordi che:

- si considerano adiacenti due celle che condividono un lato verticale o orizzontale. Non sono adiacenti le celle in diagonale. La nozione di adiacenza permette quindi di considerare la mappa come un grafo (implicito) nel quale le celle libere rappresentano i vertici, mentre le adiacenze (al massimo 4 per ogni vertice) rappresentano gli archi.
- insiemi di celle adiacenti a 2 a 2 e contenenti lo stesso valore definiscono una regione contigua
- la dimensione di una regione contigua è il numero di celle che la compongono.

Alcuni dei completamenti validi per la griglia parziale precedente sono, ad esempio:

$$\begin{pmatrix} 1 & 5 & 1 \\ 5 & 5 & 5 \\ 5 & 2 & 2 \end{pmatrix} \begin{pmatrix} 1 & 5 & 3 \\ 5 & 5 & 3 \\ 5 & 5 & 3 \end{pmatrix} \begin{pmatrix} 1 & 5 & 1 \\ 5 & 5 & 5 \\ 1 & 5 & 1 \end{pmatrix} \begin{pmatrix} 1 & 5 & 5 \\ 3 & 5 & 5 \\ 3 & 3 & 5 \end{pmatrix}$$

Si scriva un programma in C che:

- definisca una opportuna struttura dati in grado di rappresentare lo schema di gioco, utilizzabile
- per risolvere i problemi che seguono
- verifichi che una griglia già completata contenuta in un file testuale rispetti le regole del gioco dato uno schema parziale, trovi (se esiste) una soluzione ottima per il problema.

2.2.2 Struttura dati

Si definiscano una struttura dati adatta a rappresentare lo schema di gioco e si definisca la funzione di acquisizione della griglia. Il prototipo della funzione sia:

```
int** leggiMappa(FILE *fin, int *R, int *C);
```

2.2.3 Problema di verifica

Si scriva una funzione che, data una griglia di gioco già completata, verifichi se questa rispetta la regola di completamento di cui sopra. Il prototipo della funzione sia:

```
int verifica(char *fn, int **mappa);
```

2.2.4 Problema di ricerca

Si scriva una funzione ricorsiva in grado di completare uno schema parziale rispettando la regola di completamento di cui sopra, ammesso che una soluzione esista. A fronte di più soluzioni possibili, l'algoritmo prediliga soluzioni che minimizzino il numero di valori distinti usati. Il prototipo della funzione sia:

```
void solve(int **m, int R, int C);
```

3 26/01/2021

3.1 Traccia da 12 punti

3.1.1 2pt

Sia data una matrice di interi `M` (di dimensione `r` $\times$ `c`). Si scriva una funzione `change` che per ogni cella (`i`,`j`), dove `i` e `j` rappresentano rispettivamente l'indice di riga e di colonna, effettui la seguente operazione:

- Se il valore iniziale della cella (`i`,`j`) è divisibile per 3, tutta la riga `i` e la colonna `j` sono poste al valore 3

```
void change(int **M,int r,int c);
```

Esempio

$$M = \begin{pmatrix} 1 & 1 & 3 & 0 \\ 2 & 4 & 6 & 8 \\ 1 & 2 & 1 & 1 \end{pmatrix} \rightarrow M = \begin{pmatrix} 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 \\ 1 & 2 & 3 & 3 \end{pmatrix}$$

3.1.2 4pt

Definire una struttura dati adeguata a rappresentare una lista singola linkata di interi come ADT I classe, dove il tipo lista si chiami `LIST`. Indicare esplicitamente in quale modulo/file appare la definizione dei tipi proposti. **Non è ammesso l'uso di funzioni di libreria**.

Implementare una funzione caratterizzata dal seguente prototipo:

```
void f(LIST l, int *v, int n)
```

La funzione riceve una lista di interi e un vettore `v` di interi di `n` elementi. La funzione cancella dalla lista tutti i nodi contenenti un valore che non appare nel vettore `v`. **Non è ammesso l'uso di funzioni di libreria**.

Esempio

$L : 3 \rightarrow 10 \rightarrow 6 \rightarrow 3 \rightarrow 1 \rightarrow 4$
$v = \{1, 2, 3\}$

$L' : 3 \rightarrow 3 \rightarrow 1$

3.1.3 6pt

Data una funzione wrapper con il seguente prototipo

```
void solve(char *str, char **dict, int n, int s);
```

Dove `str` è una generica stringa e `dict` è un vettore di `n` stringhe distinte. Implementare la funzione wrapper `solve` e scrivere una funzione ricorsiva invocata da `solve` che determini se è possibile costruire `str` a partire dalle stringhe del dizionario, anche ripetute, usando almeno `s` stringhe diverse del dizionario. Giustificare la scelta del modello combinatorio adottato. Giustificare la scelta del/dei criteri di pruning adottati, o il motivo della loro assenza.

Esempio

```
str = "abracadabra"
n = 8
dict = {"a", "ab", "cada", "abra", "ra", "da", "ca", "bra"}
```

Risultati

```
s = 3 -> a bra cada bra
s = 4 -> a bra ca da bra
```

3.2 Traccia da 18 punti

3.2.1 Breve introduzione e definizioni

Si vuol realizzare un programma C in grado di gestire il gioco "labirinto". Un labirinto è composto da stanze collegate direttamente le une alle altre da cunicoli. I cunicoli sono bidirezionali e una data coppia di stanze può essere connessa al più da un cunicolo. Le stanze contengono ricchezze, mentre i cunicoli possono contenere trappole. La mossa elementare è attraversare un cunicolo. Se il cunicolo è dotato di trappola, questa ferisce chi vi passa. Il labirinto può quindi esser visto come un grafo non orientato, i cui i vertici rappresentano le stanze e gli archi i cunicoli. Vertici e archi sono pesati da ricchezze e da assenza/presenza di trappole rispettivamente.

Obiettivo

Un esploratore parte ad esplorare il labirinto avendo a disposizione una sua dote di punti ferita `PF` e un numero di mosse `M` e effettua un percorso nel labirinto con l'obiettivo di accumulare quante più ricchezze possibile, subendo al massimo un numero

di ferite compatibile con `PF`. Ogni trappola decrementa di 1 i `PF` dell'esploratore che passa per quel cunicolo. Ogni attraversamento di cunicolo vale una mossa.

Percorso e vincoli

- Entrata/uscita: le stanze sono caratterizzate da una profondità (un intero non negativo assegnato a priori). L'unica stanza a profondità 0 funge da ingresso e uscita dal labirinto; il percorso inizia e termina in questa stanza.
- Percorso: il percorso consiste nel passare da una stanza a un'altra utilizzando i cunicoli. Sono ammessi cicli, cioè si può passare più volte in una stanza (eccetto quella a livello 0, da cui si può solo entrare e uscire) o percorrere più volte un cunicolo.
- Interruzione: un percorso può essere interrotto prima dell'uscita, per uno tra due possibili motivi:
  - raggiungimento del massimo numero di mosse `M` (ogni attraversamento di cunicolo vale una mossa)
  - esaurimento dei punti ferita (`PF`): ogni passaggio in un cunicolo con trappola consuma 1 punto ferita (che si sottrae al valore iniziale `PF`); ci si ferma nella stanza raggiunta se il valore `PF` dell'esploratore si è azzerato nell'ultimo transito di cunicolo

Ricchezze accumulate

Ogni stanza, eccetto quella di ingresso/uscita, può contenere

- un tesoro, caratterizzato da un valore intero `V`
- una quantità di oro sparsa per la stanza, descritta dal numero di monete `O`.

L'esploratore può raccogliere un solo tesoro, mentre non ci sono limiti per l'oro: non è comunque possibile raccogliere più volte l'oro di una stanza, se vi si passa più volte. La ricchezza totale accumulata è la somma dei valori del tesoro e di tutto l'oro eventualmente raccolti (una moneta d'oro ha valore 1).

Conclusione del percorso

- Percorso normale: si raggiunge l'uscita rispettando i vincoli, l'esploratore conquista tutta la ricchezza raccolta.

- Percorso interrotto: se il percorso si è interrotto per raggiunto limite sul numero di mosse o di ferite possibili, ci si ferma nella stanza raggiunta e sono possibili due conclusioni:
  - Soccorso: se la stanza è al livello 1 oppure al 2, arrivano i soccorritori che salvano l'esploratore. I soccorritori ottengono in cambio una parte delle ricchezze accumulate: un terzo se la stanza è al livello 1, metà se al livello 2 (valori da calcolare con arrotondamento per eccesso). Questa parte si sottrae quindi alle ricchezze accumulate;
  - Fallimento: se la stanza è al livello 3 o maggiore, nessuno va a salvare l'esploratore e l'impresa fallisce.

Formato del file

Il labirinto è descritto nel file testuale `"labirinto.txt"`, con il formato seguente:

- la prima riga contiene il numero `S` di stanze
- le successive `S` righe descrivono ognuna una stanza, con formato `<nome> <profondità> <valore_tesoro> <valore_oro>` (Il nome della stanza è una stringa priva di spazi, gli altri campi sono interi non negativi)
- le successive righe (fino alla fine del file) descrivono ognuna un cunicolo, con formato `<nome_stanza_a> <nome_stanza_b> <flag_trappola>` (coppia di nomi di stanze connesse e un flag di valore 0 o 1 per indicare assenza o presenza di trappola)

Segue un esempio di file. I commenti riportati nell'esempio seguente non sono effettivamente contenuti nel file, ma servono solo a rendere più chiara la sua lettura.

```
10
// 10 Stanze totali
Ingresso 0 0 0
// Ingresso sempre a profondità 0 e privo di tesoro e oro
Stanza1 1 0 2
// Stanza1 a profondità 1 senza tesoro e 2 monete d'oro
Stanza2 3 10 3
// Stanza2 a profondità 3 con tesoro di valore 10 e 3 monete
...
// descrizione delle altre 7 stanze
Stanza1 Stanza2 0
// Un cunicolo senza trappola tra Stanza1 e Stanza2
Stanza1 Stanza3 0
// Un cunicolo senza trappola tra Stanza1 e Stanza3
Stanza1 Stanza4 1
```

```
// Un cunicolo con trappola tra Stanza1 e Stanza4
...
```

Impostazione della soluzione

Si supponga di utilizzare, per affrontare il problema, l'ADT di I classe `Graph` visto a lezione opportunamente modificato. Si realizzino le modifiche e aggiunte richieste. Il `main` non è richiesto, ma andrà aggiunto nella versione allegata alla relazione. Si assuma che i parametri `M` e `PF` siano passati direttamente come argomento al `main`.

3.2.2 Strutture dati

Definire le strutture dati necessarie per rappresentare il labirinto, cioè l'ADT di I classe `Graph`, con le eventuali aggiunte/modifiche proposte per il labirinto con stanze, cunicoli, ricchezze e trappole, suddividendo opportunamente le definizioni tra file header e sorgente. Definire anche una struttura dati `Path`, adatta a rappresentare un percorso effettuato dall'esploratore nel labirinto, comprensivo della sequenza di stanze e delle ricchezze accumulate. Acquisire il labirinto da file in una opportuna struttura dati. Scrivere la funzione:

```
Graph GRAPHload(FILE *fp);
```

La funzione di acquisizione è richiesta in quanto le parti non standard del grafo sono significative.

3.2.3 Problema di verifica

Acquisire da un secondo file un possibile percorso nel labirinto, e verificarne la congruenza. Si chiedono due funzioni:

```
Path GRAPHpathLoad(Graph g, FILE *fp);
int GRAPHpathCheck(Graph g, PATH p, int M, int PF);
```

La prima funzione acquisisce un percorso da file, che contiene unicamente la sequenza di stanze, identificate ognuna dal nome (quindi mancano ancora le ricchezze accumulate). Il parametro `g` può essere necessario o meno, a seconda di come si sia scelto di definire il tipo `Path`. La seconda funzione verifica un percorso, determinando se rispetti i vincoli: se no, la funzione ritorna 0, se sì, ritorna 1 e calcola e salva per il percorso (parametro `p`) le ricchezze complessive massime accumulabili dall'esploratore.

3.2.4 Problema di ricerca

Individuare una sequenza di mosse all'interno del labirinto (quindi un percorso) tale da massimizzare le ricchezze raccolte compatibilmente con il massimo numero di mosse possibile `M` e di punti ferita iniziali `PF`. Si scriva cioè la funzione wrapper e la corrispondente funzione ricorsiva:

```
Path GRAPHpathBest(Graph g, int M, int PF);
Path GRAPHpathBestR(Graph g, int M, int PF, ...);
```

4 16/02/2021

4.1 Traccia da 12 punti

4.1.1 2pt

Sia data una matrice di interi (positivi, negativi e/o nulli) `M` (di dimensione `r` $\times$ `c`). Si scriva una funzione `f` che identifichi e conti tutte le sottomatrici quadrate la cui somma degli elementi sia minore del massimo elemento presente nella matrice. Sia dato il seguente prototipo per la funzione:

```
int f(int **M,int r,int c);
```

4.1.2 4pt

Definire una struttura dati adeguata a rappresentare una lista singola linkata di interi come ADT I classe, dove il tipo lista si chiami `LIST`. Indicare esplicitamente in quale modulo/file appare la definizione dei tipi proposti. **Non è ammesso l'uso di funzioni di libreria**. Implementare una funzione caratterizzata dal seguente prototipo:

```
LIST* split(LIST l, int *n)
```

la funzione `split` suddivide la lista `l` in un vettore di lunghezza `n` (da determinare) di liste (valore di ritorno), dove le singole sottoliste, di lunghezza massimale, sono composte da soli elementi contigui della lista originale di valore pari/dispari. La lista originale deve rimanere inalterata. **Non è ammesso l'uso di funzioni di libreria**.

Esempio

$3 \to 10 \to 6 \to 3 \to 1 \to 5$
In output si ottiene un vettore di 3 liste, quali:
3
$10 \to 6$
$3 \to 1 \to 5$

4.1.3 6pt

Data una funzione wrapper con il seguente prototipo

```
void solve(char *start, char *end, char **dict, int n, int k);
```

Dove `start` e `end` sono due generiche stringhe e `dict` è un vettore di n stringhe distinte di soli caratteri alfabetici maiuscoli, di cui fanno parte anche `start` e `end`. Tutte le stringhe del dizionario hanno la stessa lunghezza. Implementare la funzione wrapper `solve` e scrivere una funzione ricorsiva `solveR` invocata da `solve` che determini se è possibile individuare una sequenza di stringhe appartenenti al dizionario iniziante per `start` e che termini in `end` dove due stringhe consecutive differiscono per esattamente un carattere. Ogni parola del dizionario è usabile al massimo una volta. Detta `L` la lunghezza delle parole e `i` un intero compreso tra `0` e `L-1`, i cambiamenti richiesti per l'$i-$esimo carattere, nella sequenza di parole, devono essere almeno `k`. Se ad esempio `k` valesse 2, il primo carattere (e così pure il secondo, il terzo, ecc.) deve cambiare almeno 2 volte nella sequenza.
NOTA BENE è possibile cambiare una lettera anche se coincide già con il suo valore finale in `end`. Giustificare la scelta del modello combinatorio adottato. Giustificare la scelta del/dei criteri di pruning adottati, o il motivo della loro assenza.

Esempio

Sequenza parziale di trasformazione, dove è evidenziata la singola lettera cambiata a ogni passo
. . . PALI → P**O**LI → **V**OLI → VOL**A** → **C**OLA → CO**S**A . . .

4.2 Traccia da 18 punti

4.2.1 Breve introduzione e definizioni

Una conferenza deve pianificare il programma di una giornata di presentazione dei vari articoli scientifici ricevuti dalla comunità di ricercatori.

Articoli

Gli articoli scientifici sono riportati in un file testuale `articoli.txt` organizzato come segue:

- Sulla prima riga appare il numero `A` di articoli accettati
- Seguono, in ragione di uno per riga, i dettagli di ogni articolo rappresentato da `<titolo> <relatore> <slot> <argomento>`

Il `<titolo>` di un articolo è una stringa senza spazi. Il `<relatore>` di un articolo è una stringa senza spazi, e rappresenta il nome del ricercatore incaricato di presen-

tare l'articolo. Il valore `<slot>` è un intero non negativo a rappresentare il numero di slot temporali contigui che l'articolo ha a disposizione per essere presentato. Ad esempio, il valore 1 indica un articolo a cui è concesso un solo slot, valori più alti sono assegnati ad articoli a cui sia concesso più tempo, contiguo, durante la conferenza. L'`<argomento>` è una stringa senza spazi a rappresentare il tema principale dell'articolo.

```
10
TitoloArticolo1 Relatore_A 2 TemaNum1
UnSecondoArticolo Relatore_Num2 1 TemaNum1
IlTerzoArticolo Relatore_A 3 UnAltroArgomento
...
```

Conferenza

La conferenza ha a disposizione `R` sale distinte, usate in contemporanea, e la giornata è suddivisa in `S` slot temporali. In un singolo giorno sono quindi disponibili `R`×`S` slot totali durante i quali gli articoli possono essere presentati. I valori `R` e `S` si assuma siano ricevuti dal `main` come argomenti.

Programma della giornata

Nel comporre il programma della conferenza, occorre tenere conto delle seguenti osservazioni:

- Un relatore potrebbe dover presentare più di un articolo, e queste presentazioni non possono avvenire in contemporanea
- È possibile non occupare tutti gli slot disponibili
- Tutti gli articoli devono essere presentati e disporre di tutti gli slot previsti per gli stessi

4.2.2 Strutture dati

Definire le strutture dati necessarie per rappresentare l'articolo, la collezione di articoli e il programma di una giornata. Il singolo articolo sia rappresentato mediante un tipo chiamato `ARTICOLO` implementato come quasi ADT. La collezione sia rappresentata mediante un tipo chiamato `ARTICOLI` implementato come ADT di I classe. L'implementazione della struttura dati adatta a memorizzare la pianificazione del programma della conferenza, il cui tipo è chiamato `PROGRAMMA`, è a discrezione del candidato. Indicare esplicitamente in quale modulo, file sorgente e/o di intestazione rientra quanto definito.

4.2.3 Problema di verifica

Acquisire da un secondo file un possibile programma e verificare se sia accettabile. Il formato del file è libero. Si chiede di implementare le seguenti tre funzionalità:

- Lettura di un possibile programma della conferenza
- Verifica della validità di un programma per la conferenza

Si deve garantire che il programma contenga tutte e sole le presentazioni previste e rispettare tutti i vincoli elencati in precedenza.

4.2.4 Problema di ricerca e ottimizzazione

Scrivere una funzione ricorsiva che generi, se possibile, un programma per la conferenza valido e ottimale. I criteri di ottimo sono basati su due parametri:

- Minimizzare gli slot vuoti `NV` all'inizio o nel mezzo della giornata. I buchi sarebbe meglio siano al fondo, cioè alla fine della giornata.
- Tenere articoli dello stesso tema nella stessa sala, se possibile. Si indichi con `NA_i` il numero di argomento diversi per l'$i-$esima stanza e con `NA` la sommatoria dei vari `NA_i`.

Minore sarà la somma `NV+NA`, migliore sarà il programma della conferenza.

5 15/06/2021

5.1 Traccia da 12 punti

5.1.1 2pt

Sia data una matrice quadrata di interi `M` (di dimensione `N` $\times$ `N`). Ogni cella della matrice può contenere solo il valore 0 o il valore 1. La matrice rappresenta delle relazioni di amicizia tra persone. Se in posizione `i`,`j` è contenuto il valore 1, le persone `i` e `j` sono amiche. Scrivere una funzione che individui e stampi tutte le coppie aventi esattamente `k` amici in comune

Esempio

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Coppie valide:
(0, 4) (1, 2) (1, 3) (1, 4) (1, 5) (3, 4) (3, 5) (4, 5)

5.1.2 4pt

Definire una struttura dati adeguata a rappresentare una lista singolo linkata di stringhe come ADT I classe, dove il tipo lista si chiami `LIST`. Indicare esplicitamente in quale modulo/file appare la definizione dei tipi proposti. La struttura così definita è quella da usare per completare la seconda domanda di questo esercizio. **Non è ammesso l'uso di funzioni di libreria**. Implementare una funzione caratterizzata dal seguente prototipo:

```
void split(LIST l, char *str, char sep)
```

la funzione `split` suddivide la stringa `str` in una lista di (sotto)stringhe spezzando `str` ogni volta che viene incontrato il carattere `sep`. Il carattere `sep` deve essere incluso come primo carattere della (sotto)stringa successiva a quella corrente **Attenersi al prototipo dato**.

Esempio 1

`s = "abracadabra" sep = 'r'`
$ab \rightarrow racadab \rightarrow ra$

Esempio 2

`s = "ramarro" sep = 'r'`
$rama \rightarrow r \rightarrow ro$

5.1.3 6pt

Data una funzione wrapper con il seguente prototipo

```
void solve(char *start, char *end, sub *S, int nSubs);
```

Dove `start` e `end` sono due generiche stringhe lunghe uguali. `sub` è una struttura definita come segue

```
typedef struct {
  char *s;
  int pos;
  int costo;
} sub;
```

usata per rappresentare una sostituzione. Una sostituzione è caratterizzata da una sottostringa `s`, con cui sovrascrivere una porzione di una stringa destinazione a partire dalla posizione `pos`. Ogni sostituzione ha un certo costo associato.

Esempio di sostituzione

`start = "amico" s1 = {s = "BA", pos = 1, cost = 4}`
applicando la sostituzione `s1` alla stringa `start`, si ottiene `"aBAco"`.
`s2 = {s = "TE", pos = 3, cost = 1}`
applicando una ulteriore sostituzione `s2`, si ottiene `"aBATE"`.
La trasformazione ha complessivamente costo $4 + 1 = 5$.

`S` è un vettore di sostituzioni ammissibili, di lunghezza `nSubs`. Si assuma per semplicità che tutte le sostituzioni presenti in `S` siano totalmente applicabili alla stringa `start` ricevute in input, ossia che la posizione di applicazione e la lunghezza delle sostituzioni non vada a superare la lunghezza della stringa destinazione. Scrivere una funzione ricorsiva che determini se è possibile individuare una sequenza di sostituzioni, tra quelle disponibili nel vettore `S`, in grado di trasformare la stringa `start`

nella stringa `end`, minimizzando il costo totale di trasformazione. A parità di costo prediligere sequenze di sostituzioni più lunghe.

```
void solve(char *start, char *end, sub *S, int nSubs)
```

Giustificare la scelta del modello combinatorio adottato. Giustificare la scelta del/dei criteri di pruning adottati, o il motivo della loro assenza.

Esempio di completo

```
start = "passato"
end = "persona"
nSubs = 8
S = {
  {"er", 1, 4},
  {"ers", 1, 5},
  {"sa", 3, 1},
  {"so", 3, 2},
  {"ato", 0, 1},
  {"on", 4, 2},
  {"ona", 4, 3},
  {"a", 6, 1}
}
```

`passato`
+ `er` in posizione 1 a costo 4 = `persato`
+ `on` in posizione 4 a costo 2 = `persono`
+ `a` in posizione 6 a costo 1 = `persona`
Costo complessivo di trasformazione: 7

5.2 Traccia da 18 punti

5.2.1 Breve introduzione e definizioni

Un magazzino di stoccaggio è costituito da `C` corridoi. Ogni corridoio ospita `C_s` scaffali e ogni scaffale ha a disposizione un numero complessivo fisso `K` di slot, numerati da `0` a `K-1`. Ogni corridoio è caratterizzato da un identificativo numerico unico, da `0` a `C-1`, e dal numero di scaffali che ospita. Ogni corridoio può ospitare un numero diverso di scaffali. Ogni scaffale è caratterizzato da un codice identificativo alfanumerico unico all'interno del magazzino. Si assuma che tutti gli scaffali del magazzino abbiano il medesimo numero di slot. Il magazzino è utilizzato per conservare dei pacchi. Ogni pacco è caratterizzato da un codice alfanumerico unico.

Per ogni pacco si è interessati a memorizzare la posizione dello stesso nel magazzino, in termini della terna `<corridoio, scaffale, slot>`. Si assuma che tutti i pacchi siano di egual dimensione, e occupino un singolo slot ciascuno.

5.2.2 Strutture dati

Definire delle opportune strutture dati così da poter rappresentare lo scenario precedentemente descritto, rispettando i seguenti vincoli:

- tipo `Pacco`: implementazione a discrezione del candidato
- tipo `Scaffale`: implementazione a discrezione del candidato
- tipo `Corridoio`: collezione di `Scaffale` (ADT I categoria)
- tipo `Magazzino`: collezione di `Corridoio` (ADT I categoria)

La base dati deve supportare le seguenti operazioni:

- Ricerca della posizione di un pacco, dato il codice, **con complessità al più logaritmica nel numero dei pacchi totali del magazzino**. Per questo punto, in caso si voglia usare un *BST*, si assuma che siano rispettate automaticamente le proprietà di bilanciamento. In caso si faccia uso di strutture dati di libreria, indicare esplicitamente la tipologia di "*Item*" contenuto e l'eventuale definizione del nodo contenitore;
- Inserimento di un pacco in una data posizione, con eventuale indicazione di errore a fronte di posizione già occupata/non esistente;
- Estrazione di un pacco da una data posizione, con eventuale indicazione di errore a fronte di posizione vuota/non esistente;
- Spostamento di un pacco da una posizione a un'altra, eventualmente in un corridoio/scaffale diverso, con gestione degli errori di cui sopra;
- Compattazione dei contenuti di due scaffali in uno solo dei due, se possibile

Organizzare la soluzione suddividendo il codice in più moduli, uno per ogni tipo richiesto, e un client principale.

6 01/09/2021

6.1 Traccia da 12 punti

6.1.1 2pt

Siano dati due vettori di interi positivi `v1` e `v2` di dimensioni `d1` e `d2`. Si scriva una funzione in grado di generare una matrice di dimensioni `d1` $\times$ `d2` in cui il contenuto della generica cella `[i,j]` sia il prodotto dell'elemento di posizione `i` di `v1` e dell'elemento di posizione `j` di `v2` Il prototipo della funzione sia quello presentato a seguire. L'allocazione della matrice deve avvenire dentro alla funzione. La matrice creata e i suoi contenuti devono essere disponili per chi invoca la funzione. Il prototipo va completato opportunamente in modo da poter rispondere alle specifiche della richiesta.

```
void f(int *v1, int *v2, int d1, int d2, ...);
```

Esempio

$v1 = \{2, 4, 6\}$
$v2 = \{1, 3, 5, 7\}$
$$M = \begin{pmatrix} 2 & 6 & 10 & 14 \\ 4 & 12 & 20 & 28 \\ 6 & 18 & 30 & 42 \end{pmatrix}$$

6.1.2 4pt

Definire una struttura dati adeguata a rappresentare una lista doppio linkata di interi come ADT I classe, dove il tipo lista si chiami `LIST`. **La lista definita non deve fare uso di sentinelle**.
Indicare esplicitamente in quale modulo/file appare la definizione dei tipi proposti. La struttura così definita è quella da usare per completare la seconda domanda di questo esercizio. **Non è ammesso l'uso di funzioni di libreria**.
Implementare una funzione caratterizzata dal seguente prototipo:

```
void purge(LIST l, int div)
```

la funzione `purge` cancella dalla lista tutti i nodi che contengono un numero non divisibile per `div`.
Non è ammesso l'uso di funzioni di libreria. Attenersi al prototipo dato.

Esempio

`div = 3`
$L = 9 \leftrightarrow 8 \leftrightarrow 7 \leftrightarrow 6 \leftrightarrow 5 \leftrightarrow 4 \leftrightarrow 3 \leftrightarrow 2 \leftrightarrow 1 \leftrightarrow 0$

$L' = 9 \leftrightarrow 6 \leftrightarrow 3 \leftrightarrow 0$

6.1.3 6pt

Data una funzione wrapper con il seguente prototipo

```
void solve(char *target, part *P, int nParts);
```

Dove `target` è una stringa obiettivo da costruire sfruttando le stringhe disponibili nel vettore di strutture `P`, di tipo `part`, lungo `nParts`. `part` è una struttura definita come segue:

```
typedef struct {
  char *s;
  int pos;
  int costo;
} part;
```

usata per rappresentare una possibile *parte* della stringa obiettivo. Una parte è caratterizzata da una sottostringa `s`, che può essere posizionata solo a partire dal carattere di posizione `pos`. Ogni parte ha un certo costo associato. Scrivere una funzione ricorsiva che determini se è possibile generare la stringa `target` sfruttando le stringhe disponibili in `P` minimizzando il costo totale di costruzione. A parità di costo prediligere la soluzione che fa uso di più parti. Giustificare la scelta del modello combinatorio adottato. Giustificare la scelta del/dei criteri di pruning adottati, o il motivo della loro assenza.

Esempio

```
target = "persona"
P = {
  {"p", 0, 1}, // p0
  {"pers", 0, 5}, // p1
  {"er", 1, 4}, // p2
  {"ers", 1, 4}, // p3
  {"sa", 3, 1}, // p4
```

```
  {"so", 3, 2}, // p5
  {"ato", 0, 1}, // p6
  {"on", 4, 2}, // p7
  {"ona", 4, 3}, // p8
  {"a", 6, 1} // p9
}
```

Usando `p0`, `p3`, `p7`, `p9`: costo 8, cardinalità soluzione 4 (soluzione ottima).
Usando `p0`, `p3`, `p8`: costo 8, cardinalità soluzione 3.
Usando `p1`, `p8`: costo 8, cardinalità soluzione 2.

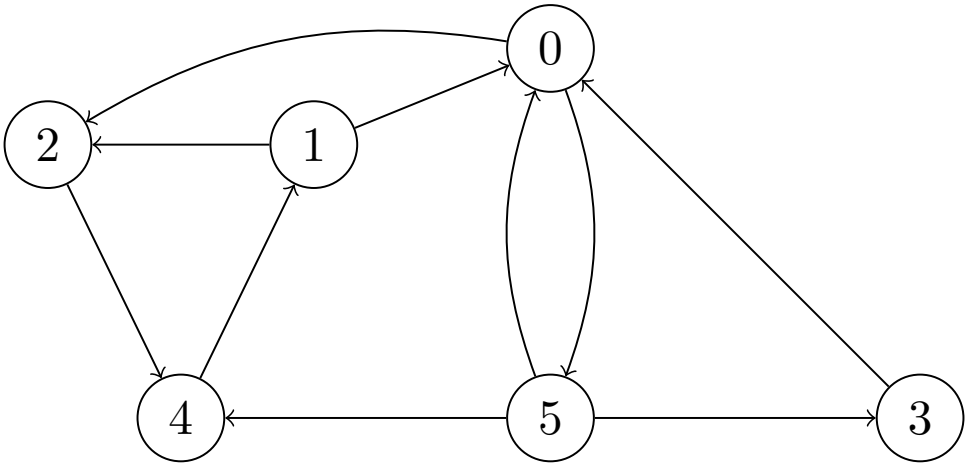
6.2 Traccia da 18 punti

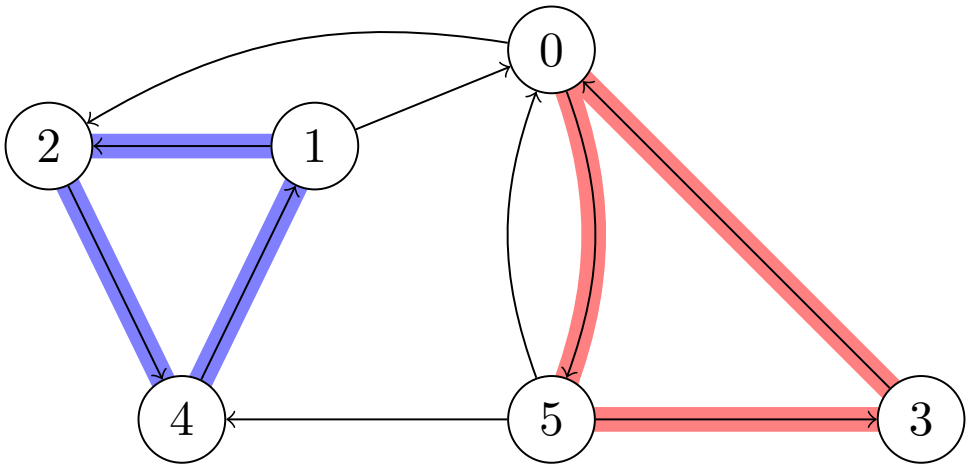
6.2.1 Breve introduzione e definizioni

Dato un grafo $G = (V, E)$ orientato e connesso, si definisce *copertura ciclica* un insieme di cicli la cui unione copre tutti i vertici del grafo. Perché un vertice sia ritenuto coperto, deve quindi far parte di almeno uno dei cicli inclusi nell'insieme preso in considerazione. Nel contesto dell'appello corrente, quando di parla di cicli, si intende sempre e solo **cicli semplici**. Una copertura può essere *sovrapposta*, se lo stesso vertice appare in più cicli, o *disgiunta*, nel caso contrario.

Esempio

A seguire è riportato un grafo di esempio e una sua copertura disgiunta che fa uso di due cicli.





Il grafo è memorizzato in un file testuale `grafo.txt` organizzato come segue:

- Sulla prima riga appare una coppia di interi `V` e `E` a rappresentare il numero di vertici e archi del grafo;
- Seguono `E` righe, ognuna della quale riporta una coppia `v w` a rappresentare i singoli archi del grafo;
- Si assuma che i vertici siano identificati univocamente con un valore intero nell'intervallo `0` ... `V-1`.

L'insieme dei cicli è anch'esso fornito, ed è memorizzato in un file testuale `cicli.txt` organizzato come segue:

- I cicli sono riportati in ragione di uno per riga. Il numero di cicli non è noto a priori;
- Ogni $i-$esima riga presenta un primo valore intero `len_i`, rappresentante la lunghezza del ciclo, seguito da `len_i` valori interi a rappresentare i vertici che compongono il ciclo stesso.

A seguire sono riportati due file completi che riprendono i contenuti dell'esempio presentato in precedenza.

`grafo.txt`

```
6 10
0 2
0 5
1 0
1 2
2 4
```

```
3 0
4 1
5 0
5 3
5 4
```

`cicli.txt`

```
4 0 5 4 1
3 0 5 3
2 0 5
4 0 2 4 1
3 1 2 4
```

6.2.2 Strutture dati

Acquisire in una opportuna struttura dati idonea i contenuti del file `grafo.txt`. Se si fa uso della struttura grafo ADT vista a lezione, riportare le eventuali modifiche apportate alla sua definizione. Definire una struttura dati idonea a memorizzare un numero arbitrario di cicli e implementare le funzioni opportune per acquisire i contenuti del file `cicli.txt`.

6.2.3 Problema di verifica

Dato un elenco ordinato di vertici, in un formato a discrezione del candidato, verificare se questo rappresenta un ciclo.

6.2.4 Problema di ricerca e ottimizzazione

Individuare, se possibile, un insieme a cardinalità minima di cicli, scelti dall'elenco letto in precedenza, la cui unione copre tutti i vertici del grafo. A parità di cardinalità, prediligere una copertura disgiunta.