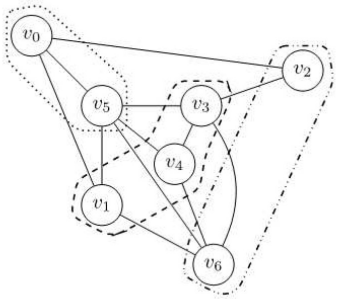
DOMATIC PARTITION PER UN GRAFO NON ORIENTATO

Dato un grafo G=(V,E) non orientato e non pesato, viene definita **Domatic Partition** una partizione $\{ V_1, V_2, .., V_k\}$ dei vertici del grafo tale per cui ogni sottoinsieme V è un **Dominating Set**.
Un set D di vertici è definito **Dominating**, se ogni vertice NON in D è adiacente ad almeno un vertice di D.



La soluzione proposta è un programma che permette di calcolare la Domatic Partition con cardinalità maggiore, quindi con il maggior numero di Dominating Set, che viene salvata sul file **proposta.txt** .

MODULI E STRUTTURE DATI UTILIZZATI
Il programma è modulare e utilizza i seguenti moduli:
1)modulo Graph
con file header .h e file di implementazione .c, con le seguenti funzioni:

Graph GRAPHinit(int V)
inizializza la struttura dati utilizzata per salvare il grafo
void GRAPHfree(Graph G)
distrugge la struttura dati del grafo e libera la memoria
Graph GRAPHreadFile(FILE *fin)
legge il file **g.txt** e salva il grafo in una struttura dati opportuna
void GRAPHinsertE(Graph G, int id1, int id2)
dati due vertici, crea l'arco associato e lo aggiunge alla struttura del grafo
int GRAPHgetIndex(Graph G, char *label)
ritorna l'indice del vertice, dato il nome
char* GRAPHVertexname(Graph G,int i)
ritorna il nome del vertice, dato l'indice
int checkDominatingSet(Graph G, int *set)
controlla che il set di vertici passato verifichi la proprietà di dominating

La struttura dati è stata leggermente cambiata rispetto a quella proposta in sede d'esame, poiché si) abbandona la scelta di utilizzare le **liste di adiacenza** , prediligendo invece una **matrice di adiacenze**.
Questa scelta è funzionale alla funzione di verifica **checkDominatingSet**, perché anziché scorrere le liste di adiacenza è sufficiente un accesso diretto alla matrice di adiacenza nella posizione (i,j), per controllare la presenza dell'arco tra il vertice v_i non appartenente a D e il vertice v_j appartenente a D.

Il modulo Graph è utilizzatore del
2)modulo ST (symboltable)
con file header .h e file di implementazione .c, con le seguenti funzioni:

ST STinit(int maxN)
inizializza la struttura dati per la symboltable
void STfree(ST st)
libera la struttura dati
int STinsert(ST st, char *s)
inserisce il nome del vertice all'indice opportuno
int STsearch(ST st, char *s)
dato il nome del vertice, ritorna l'indice
int STcount(ST st)
ritorna il numero di vertici inserito fin ora
char *STsearchByIndex(ST st, int i)
dato l'indice, ritorna il nome del vertice

La presenza di una symboltable ST è superflua , perché in questo caso non serve sapere il nome dei vertici ed è possibile lavorare soltanto sugli indici. Si sceglie di implementarla lo stesso perché costa poco in termini di tempo e di memoria, ma c'è il vantaggio di poter aggiornare ed estendere il programma con più facilità, tenendo conto di eventuali richieste future.

FILE DI INPUT E OUTPUT
L'unica funzione del modulo ST è il salvataggio dei nodi, del loro nome e degli archi associati, infatti il programma si avvale di due file di testo:

1)**g.txt**, file di input con il seguente formato
· sulla prima riga compare il numero di vertici V
· segue un numero non definito di coppie v w a rappresentare gli archi del grafo, con v e w compresi tra 0 e V-1
Se considero l'esempio in figura , il file g.txt risulta

```
- - - - - - - - - - - - - - - -
|7                      |
|v0 v2                  |
|v0 v1                  |
|v0 v5                  |
|v2 v3                  |
|v3 v5                  |
|v3 v4                  |
|v4 v5                  |
|v5 v1                  |
|v5 v6                  |
|v3 v6                  |
|v4 v6                  |
|v1 v6                  |
- - - - - - - - - - - - - - - -
```

2)**proposta.txt**, file di output con il seguente formato
V interi separati da uno spazio, con valore da 0 a V-1
indicano a quale partizione appartiene il vertice il cui indice corrisponde all'ordine
Se considero la Domatic Partition dell'esempio, il file proposta.txt risulta

```
- - - - - - - - - -
|0 1 2 1 1 0 2 |
- - - - - - - - - -
```

VERIFICA DELLA SOLUZIONE E ALGORITMO RICORSIVO
Nel file main.c, viene lasciata l'implementazione della funzione
int checkPartition(Graph G, FILE *fin)
-riceve il grafo G e il file proposta.txt , verifica che la partizione sia Domatic
-wrapper della funzione
int checkDomaticPartition(Graph G,int *sol, int k)
riceve il grafo G, un vettore di interi che contiene la proposta, la cardinalità del partizionamento

Si osserva che la funzione **checkPartition()** sarebbe molto utile solo nel caso in cui si scegliesse di ricorrere alle disposizioni ripetute per generare una possibile soluzione, per poi controllarne la validità.
Bisognerebbe quindi aggiornare il massimo corrente ogni volta, per trovare infine la Domatic Partition a cardinalità massima.

Questo è corretto dal punto di vista risolutivo, tuttavia il modello scelto per l'algoritmo ricorsivo è quello delle partizioni, anche detto **Algoritmo di ER**. È possibile sfruttare questo modello in modo molto più efficiente, si può quindi fare a meno della funzione **checkPartition()** e sfruttare soltanto la funzione di verifica **checkDomaticPartition()** all'interno della funzione

int generaPartizioni(Graph G, int n, int m, int pos, int *sol, int k)
-funzione ricorsiva sul modello dell'Algoritmo di ER
-riceve il grafo G, il numero di elementi n, il numero m di blocchi (compreso tra 1 e n)
-genera tutte le possibili partizioni, termina l'esecuzione appena ne trova una che sia Domatic
-salva la soluzione nel vettore *sol

void generaDomatic(Graph G)
-wrapper della funzione **generaPartizioni()**
-contiene un ciclo for che permette di calcolare la partizione con cardinalità k nota a priori
-salva la partizione trovata nel file **proposta.txt**
-il valore della cardinalità k decresce a partire da V fino a 0

Questo approccio consente di terminare l'esecuzione alla prima partizione Domatic trovata: il flusso di esecuzione è a cardinalità decrescente, quindi la prima soluzione trovata è anche quella con cardinalità massima.