Liste di adiacenza con vertici memorizzati in array

Esame 23-7

0 2 0
0 0 3
0 0 0

La matrice rappresenta un grafo orientato con **tre vertici** numerati come 0, 1 e 2. Questi numeri corrispondono alle **righe e colonne della matrice.**
La cella **(i, j)** della matrice indica se c'è un arco dal vertice **i** al vertice **j.**
Per esempio (0, 1) = 2 significa che esiste un arco dal vertice 0 al vertice 1 con **peso** 2.

La prima funzione da implementare per questo esame è quello di calcolarmi il peso totale del grafo. Nel grafo, il peso è un valore numerico associato agli archi. Quindi per esempio il peso tra 0 e 1 è 2.

Codice:

```
int pesoTotaleArchi(const Grafo& g) {
  int sum = 0;
  for (int i = 0; i < g.numVertici; ++i) {
    for (int j = 0; j < g.numVertici; ++j) {
      if (g.matrice[i][j] != 0) {
        sum += g.matrice[i][j];
      }
    }
  }
  return sum;
}
```

La seconda funzione invece mi indica se esiste un vertice intermedio tra due vertici dati.
Per esempio se v1 vale 1 e v2 vale 0 la logica è la seguente:

Se **v1** è l'indice della riga e **v2** è l'indice della colonna, devo verificare se esiste un **vertice intermedio i** tale che ci sia un arco da **v1** a **i** (ovvero **g[v1][i] != 0**) e un arco da **i** a **v2** (ovvero **g[i][v2] != 0**). Se entrambe queste condizioni sono vere, allora il vertice **i** è un vertice intermedio che collega **v1** e **v2,** quindi aggiorno vlink a **i** e restituisco **true**. Se non trovo alcun vertice intermedio che soddisfi entrambe le condizioni, restituisco **false** e assegno **vlink** al valore elementoInesistente
Codice:s

```
bool sonoCollegatiDaUnVertice(const Grafo& g, VerticeID v1, VerticeID v2, VerticeID& vlink)
{

   for (int i = 0; i < g.numVertici; ++i) {
      if (g.matrice[v1][i] != 0 && g.matrice[i][v2] != 0) {
         vlink = i;
         return true;
      }
   }
   vlink= elementoInesistente;
   return false;
}
```

Esame 2021

Ci viene chiesto di trovare il grado minore del vertice. Inizializziamo una variabile **minGrado** con un valore alto e una variabile **verticeMinimo** a -1. Abbiamo un primo ciclo for per poter scorrere le righe, dentro a questo for inizializziamo una variabile **gradoUscente.** Scorriamo le colonne con un secondo **ciclo for** e all'interno di esso controlliamo se in posizione **[i][j]** l'arco esiste. In caso affermativo incrementiamo il **gradoUscente.** All'interno del ciclo for principale controlliamo se **gradoUscente** sia minore di **minGrado.** In caso positivo **minGrado** prende il valore di **gradoUscente** e salviamo la **posizione i-esima** dentro a **verticeMinimo.**

Codice:

```
int ritornaVerticeAventeGradoMin(const Grafo& g) {
   int minGrado;  // Inizializzo il grado minimo a un valore molto alto
   int verticeMinimo = -1;        // Variabile per memorizzare il vertice con il grado
minimo

   for (int i = 0; i < g.numVertici; ++i) {
      int gradoUscente = 0;
      for (int j = 0; j < g.numVertici; ++j) {
         if (g.matrice[i][j] != 0) {
            gradoUscente++;
         }
      }

      // Se il grado uscente è inferiore al minimo trovato finora, aggiorna il minimo
      if (gradoUscente < minGrado) {
         minGrado = gradoUscente;
         verticeMinimo = i;
      }
   }

   return verticeMinimo;
}
```

La funzione **camminoNelGrafoDecrescente** verifica se nel **grafo g** esiste un cammino rappresentato **dall'array C** in cui i pesi degli archi sono **strettamente decrescenti.**
Iniziamo a scorrere mediante un ciclo for i nodi del **cammino C,** la variabile **l** rappresenta la **lunghezza di C. La prima if** riguarda il primo arco, inizializziamo peso con il valore del primo arco tra **C[0] e C[1].** Mentre la seconda controlla se il peso successivo è maggiore al precedente, se risultasse positivo allora non è strettamente decrescente e ritorniamo false.

Codice

```
bool camminoNelGrafoDecrescente(const Grafo& g, const int& l, const Cammino C) {
  if (g.numVertici <= 0) {
    return false;
  }

  if (l < 1) {
    cout << "Cammino corto";
    return false;
  }

  int peso;
  for (int i = 0; i < l - 1; ++i) {
    if (i == 0) {
        peso = g.matrice[C[i]][C[i + 1]];
    } else {
        if (peso <= g.matrice[C[i]][C[i + 1]]) {
            return false;
        }
        peso = g.matrice[C[i]][C[i + 1]];
    }
  }
  return true;
}
```

La funzione **contaSink** controlla se il vertice corrente è un sink, ovvero se non ha archi uscenti. Usiamo un contatore e semplicemente scorriamo tutto il grafo, Se in posizione **[i][j]** abbiamo un numero != 0 allora usciamo dal ciclo

Codice

```
int contaSink(Grafo& g) {
   int count = 0;
   for (int i = 0; i < g.numVertici; ++i) {
      bool isSink = true; // Presupponiamo che il vertice sia un sink
      for (int j = 0; j < g.numVertici; ++j) {
         if (g.matrice[i][j] != 0) {
            isSink = false;
            break;
         }
      }
```

```
        // Se il vertice è un sink, incrementa il contatore
        if (isSink) {
            count++;
        }
    }

    return count;
}
```