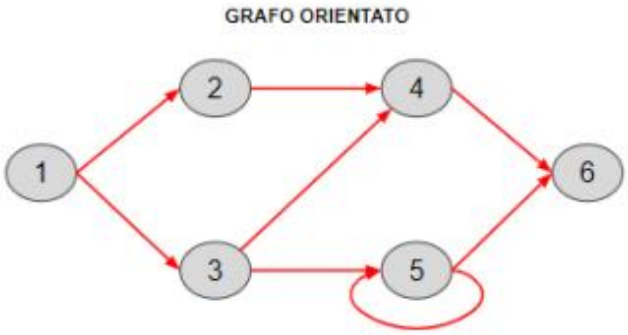
PROBJOT SINGH 869434
P.singh3@campus.unimib.it

PROGETTO D'ESAME DI GIUGNO – GRAFO



nodi destinazione

nodi di partenza	1	2	3	4	5	6
1	0	1	1	0	0	0
2	0	0	0	1	0	0
3	0	0	0	1	1	0
4	0	0	0	0	0	1
5	0	0	0	0	1	1
6	0	0	0	0	0	0

PROGETTAZIONE/IMPLEMENTAZIONE:

Il progetto richiede la progettazione e realizzazione di una classe generica che implementa un grafo orientato. Un **grafo** è costituito da un insieme di nodi e archi. I nodi sono rappresentati da un generico identificativo (es. un numero, una stringa, un oggetto, ecc...). Gli archi mettono in relazione due nodi creando un collegamento tra loro. Gli archi sono orientati.
Il grafo è stato implementato mediante una matrice di adiacenza V x V, dove V è il numero di vertici(nodi) del grafo. La matrice, booleana, serve ad identificare la presenza o meno di un arco tra il nodo i e il nodo j. Essendo il grafo orientato, la matrice non sarà simmetrica, quindi avere un arco da i a j non significa averne uno che da j ad i.
È stato utilizzato un array, anch'esso allocato dinamicamente, di supporto per la gestione degli identificativi dei nodi, si tratta di un array parallelo, alle colonne della matrice booleana, di conseguenza l'elemento in posizione i dell'array, sarà l'identificativo del nodo i/j della matrice (equivale anche alla i-esima riga, o alla i-esima colonna).

Identificativo nodo

1	2	3	4	5	6

nodi destinazione

nodi di partenza	1	2	3	4	5	6
1	0	1	1	0	0	0
2	0	0	0	1	0	0
3	0	0	0	1	1	0
4	0	0	0	0	0	1
5	0	0	0	0	1	1
6	0	0	0	0	0	0

PROBJOT SINGH 869434
P.singh3@campus.unimib.it

Inoltre si tratta di una classe templata, poiché gli identificativi dei nodi possono essere di un tipo qualsiasi.

```
private:
    T* _idArray; //pointer to the support array for the identificators
    bool** _adjMatrix; //pointer to the adjency matrix
    size_type _aSize; //size of the array
    int indexOf(const T &value)const{ //returns the index of the parameter ID in the array
        int index = -1;
        for (size_type i = 0; i < _aSize; i++)
            if (_idArray[i] == value)
                index = i;
        return index;
    }
}; //END CLASS GRAPH
```

Nelle variabili private abbiamo quindi, il puntatore all'array parallelo, il puntatore alla matrice booleana, una variabile **_aSize**, che indica la dimensione dell'array parallelo, di conseguenza anche il numero di nodi del grafo, infatti verrà usato anche per istanziare la matrice V x V. Inoltre abbiamo un metodo privato **indexOf(const T&)** che ritorna l'indice del nodo che gli viene passato. Si noti che la matrice è stata implementata con un doppio puntatore, e verrà poi allocata dinamicamente come un array di puntatori a bool, i cui elementi a loro volta sono array.

COSTRUTTORE / DISTRUTTORE

```
graph() : _idArray(nullptr), _adjMatrix(nullptr), _aSize(0) {
    #ifndef NDEBUG
        std::cout << "graph::graph()" << std::endl;
    #endif
}
```

È stato implementato il solo **costruttore di default**, il quale crea un grafo vuoto a cui poi verranno aggiungi i nodi dall'utente mediante i metodi a disposizione. Questo costruttore setta i puntatori a **nullptr** be la dimensione a zero. È stato scelto di non implementare nessun costruttore secondario, poiché quest'ultimo avrebbe chiesto l'intervento dell'utente per la scelta degli indentificativi dei nodi, e anche se questi fossero stati generati casualmente, l'utente che utilizza la classe non ne sarebbe a conoscenza. È risultato più logico il solo costruttore di default.

```
graph(const graph& other) : _idArray(nullptr), _adjMatrix(nullptr), _aSize(0) {
    _idArray = new T[other._aSize];
    _aSize = other._aSize;
    _adjMatrix = new bool*[other._aSize];
    for (int i = 0; i < other._aSize; i++)
        _adjMatrix[i] = new bool[other._aSize];
    try {
        //fill the parallel array
        for (size_type i = 0; i < other._aSize; i++)
            this->_idArray[i] = other._idArray[i];
        //fill the matrix
        for (size_type i = 0; i < other._aSize; i++)
            for (size_type j = 0; j < other._aSize; j++)
            this->_adjMatrix[i][j] = other._adjMatrix[i][j];
    }
    catch (...) {
        delete[] _idArray;
        delete[] _adjMatrix;
        _idArray = nullptr;
        _adjMatrix = nullptr;
        _aSize = 0;
        throw; // rilancio dell'eccezione !!
    }
    #ifndef NDEBUG
        std::cout << "graph::graph(const graph &)" << std::endl;
    #endif
}
```

PROBJOT SINGH 869434
P.singh3@campus.unimib.it

È stato implementato un **copy constructor**, che prende in input un altro oggetto grafo, e ne crea una copia, se questo è fattibile, altrimenti viene lanciata un'eccezione, e vengono ripristinate le variabili private. Si fa notare anche qui la scelta sull'allocazione della matrice, citata sopra.

```
~graph() {
    delete[] _idArray;
    delete[] _adjMatrix;
    _idArray = nullptr;
    _adjMatrix = nullptr;
    _aSize = 0;

    #ifndef NDEBUG
        std::cout << "graph::~graph()" << std::endl;
    #endif
}
```

Il **distruttore** dealloca la memoria dallo heap, eliminando ricorsivamente l'array, la matrice, e i loro componenti.

ALTRI METODI FONDAMENTALI

```
graph& operator=(const graph& other) {
    if (this != &other) {

        graph tmp(other);

        this->swap(tmp);
    }

    #ifndef NDEBUG
        std::cout << "graph::operator=(const graph &)" << std::endl;
    #endif

    return *this;
}
```

```
void swap(graph& other) {
    std::swap(this->_idArray, other._idArray);
    std::swap(this->_adjMatrix, other._adjMatrix);
    std::swap(this->_aSize, other._aSize);
}
```

Overload dell'operatore = che utilizzando il supporto del copy constructor, crea un altro grafo uguale e lo assegna. Utilizzato come supporto anche il metodo swap, ridefinito sull'oggetto grafo.

```
bool operator == (const graph& other) {
    if (this->_aSize != other._aSize)
        return false;
    for (size_type i = 0; i < _aSize; i++)
        if (this->_idArray[i] != other._idArray[i])
            return false;
    for (size_type i = 0; i < other._aSize; i++)
        for (size_type j = 0; j < other._aSize; j++)
            if (this->_adjMatrix[i][j] != other._adjMatrix[i][j])
                return false;
    #ifndef NDEBUG
        std::cout << "graph::operator==(const graph &)" << std::endl;
    #endif
    return true;
}
```

Overload dell'operatore == che confronta due grafi, e ne verifica l'uguaglianza. Due grafi sono uguali se hanno lo stesso numero di nodi, gli stessi nodi(etichette) e gli stessi archi.

PROBJOT SINGH 869434
P.singh3@campus.unimib.it

```cpp
friend std::ostream& operator<<(std::ostream& os, const graph<T>& m) {
    for (typename graph<T>::size_type i = 0; i < m.getNodes(); i++) {
        os << m._idArray[i] << ": ";
        for (typename graph<T>::size_type j = 0; j < m.getNodes(); j++) {
            if (m._adjMatrix[i][j] == true)
                os << m._idArray[j] << ", ";
            if (j == m.getNodes() - 1 && i != m.getNodes() - 1)
                os << std::endl;
        }
    }
    return os;
}
```

Ridefinizione dell'operatore di stream, che stampa un grafo, ed è stato implementato in modo da rendere possibile anche la serializzazione, senza inserire troppi "endl" per estetica. Viene stampato il nodo, in ordine di inserimento, e dopo i " : " vengono listati i collegamenti a tale nodo. Fornisco un esempio di stampa:

```
******* stampa dopo inserimento archi *******
1: 2, 3,
2: 4, 6,
3: 4, 5,
4: 6,
5: 5,
6:
```

METODI FONDAMENTALI DI UN GRAFO

```cpp
void add_node(const T &node) {
    if(exists(node))
        throw std::invalid_argument("node already exists!");
    else {
        T* tmpArray = new T[_aSize + 1]; //allocate new array
        bool** tmpMatrix = new bool*[_aSize + 1];//allocate new matrix
        for (int i = 0; i < _aSize + 1; i++) {
            // Declare a memory block
            // of size _aSize
            tmpMatrix[i] = new bool[_aSize + 1];
        }
        //filling the new idArray
        for (size_type i = 0; i < _aSize; i++)
            tmpArray[i] = _idArray[i];

        tmpArray[_aSize] = node; //inserting new node

        //filling the new adjency matrix
        for (size_type i = 0; i < _aSize; i++)
            for (size_type j = 0; j < _aSize; j++)
                tmpMatrix[i][j] = _adjMatrix[i][j];

        for (size_type i = 0; i < _aSize + 1; i++){
            tmpMatrix[_aSize][i] = false;  //new isolated node
            tmpMatrix[i][_aSize] = false;
        }
        //make the tmps the new objects
        std::swap(_idArray, tmpArray);
        std::swap(_adjMatrix, tmpMatrix);
        this->_aSize++;
        //remove from heap the old allocated memory
        delete[] tmpArray;
        delete[] tmpMatrix;
        #ifndef NDEBUG
            std::cout << "ADDING NEW NODE : " << node << std::endl;
        #endif
    }
}
```

PROBJOT SINGH 869434
P.singh3@campus.unimib.it

Metodo per **aggiungere un nuovo nodo** al grafo. Dato che la dimensione delle strutture dati coinvolte non è modificabile(estendibile) ma fissa, vengono create due nuove strutture di supporto, di dimensione incrementata di uno rispetto quella precedente, poiché stiamo aggiungendo un nuovo nodo. Successivamente vengono copiati i vecchi dati nelle nuove strutture dati, e infine, nel caso dell'array nel nuovo spazio creato, cioè in nell'ultima posizione, viene inserito il nodo, passato come parametro, invece nel caso della matrice di adiacenza, la nuova riga e la nuova colonna vengono settate a false, poiché è richiesto che il nuovo nodo aggiunto sia isolato. Alle variabili originali vengono assegnate le strutture di supporto, tramite swap, e rimuovendo quelle con i vecchi dati dallo heap.

```
void add_arc(const T &node1,const T &node2) {
    if (connected(node1, node2)) //connected also control if they exist
        throw std::invalid_argument("nodes already connected!");
    else
        _adjMatrix[indexOf(node1)][indexOf(node2)] = true;
}
```

Metodo per **aggiungere un nuovo arco** al grafo. Viene effettuato un controllo sfruttando il metodo connected, che restituisce true se i due nodi passati in input esistono e sono connessi, cioè se esite un arco che li collega. Se ciò è vero, non è possibile inserire un arco poiché esso esiste già, e viene lanciata un'eccezione. Se non esiste già un arco, questo viene creato cercando nell'array parallelo la posizione dei due nodi, e settando a true la cella in quella posizione nella matrice.

```
void remove_node(const T &node) {
    if(!exists(node))
        throw std::invalid_argument("node does not exist!");
    else {
        T* tmpArray = new T[_aSize - 1];
        bool** tmpMatrix = new bool* [_aSize - 1];
        for (int i = 0; i < _aSize - 1; i++) {

            // Declare a memory block
            // of size _aSize
            tmpMatrix[i] = new bool[_aSize - 1];
        }

        //filling the new idArray
        for (size_type i = 0, j = 0; i < _aSize; i++)
            if (i != indexOf(node)) {
                tmpArray[j] = _idArray[i];
                j++;
            }

        //filling the new adjency matrix
        for (size_type i = 0, k = 0; i < _aSize; i++){
            if (i != indexOf(node)) {
                for (size_type j = 0, l = 0; j < _aSize; j++)
                                    if (j != indexOf(node)) {
                                        tmpMatrix[k][l] = _adjMatrix[i][j];
                                        l++;
                                    }

                k++;
            }
        }
```

PROBJOT SINGH 869434
P.singh3@campus.unimib.it

```
        //make the tmps the new objects
        std::swap(tmpArray, _idArray);
        std::swap(tmpMatrix, _adjMatrix);
        this->_aSize--;
        //remove from heap the old allocated memory
        delete[] tmpArray;
        delete[] tmpMatrix;

        #ifndef NDEBUG
            std::cout << "REMOVING THE NODE : " << node << std::endl;
        #endif

    }
}
```

Metodo per **rimuovere un nodo** dal grafo. Prima di tutto viene effettuato un controllo se il nodo passato come parametro esiste. Se questo non esiste, viene lanciata un'eccezione poiché non si può rimuovere un nodo che non c'è. Se esite, invece, vengono create le nuove strutture di supporto, di dimensione ridotta di uno, e queste vengono riempite con i dati delle vecchie strutture, EVITANDO la posizione in cui è presente un dato del nodo che stiamo rimuovendo. Create le nuove strutture, queste vengono rese le strutture attuali, facendo swap, e rimuovendo dallo heap quelle contenenti i vecchi dati.

```
void remove_arc(const T &node1,const T &node2) {
    if (!connected(node1, node2)) //connected also control if they exist
        throw std::invalid_argument("arc does not exist!");
    else
        _adjMatrix[indexOf(node1)][indexOf(node2)] = false;
}
```

Metodo per **rimuovere un arco** dal grafo. Viene anche qui utilizzato il metodo connected, per verificare se esiste un arco che collega i due nodi, in caso di esistano. Se tale arco NON esiste, viene lanciata un'eccezione, poiché non possiamo rimuovere un arco che non esiste. In caso l'arco esista, questo viene rimosso settando a false la cella corrispondente.

RICHIESTE SODDISFATTE

```
bool exists(const T &node) const{
    bool ok = false;
    for (size_type i = 0; i < _aSize; i++)
        if (_idArray[i] == node)
            ok = true;
    return ok;
}
```

È stato implementato un metodo **exists()** che ritorna true se un nodo esiste. Lo fa controllando nell'array parallelo contenente i nodi, confrontando ogni nodo con quello passato in input.

PROBJOT SINGH 869434
P.singh3@campus.unimib.it

```cpp
bool connected(const T &node1,const T &node2) const{
    if(exists(node1) && exists(node2))
        return _adjMatrix[indexOf(node1)][indexOf(node2)];
    else throw std::invalid_argument("an input node does not exist!");
}
```

Metodo **connected()** che ritorna true se i nodi passati in input sono connessi tramite un arco diretto, se i nodi esistono, altrimenti lancia un'eccezione.

```cpp
/**
@brief A const-iterator that iterates on the nodes' IDs.
*/
class const_iterator {
public:
    typedef std::forward_iterator_tag iterator_category;
    typedef T                         value_type;
    typedef ptrdiff_t                 difference_type;
    typedef const T* pointer;
    typedef const T& reference;


    const_iterator() {
        ptr = nullptr;
    }

    const_iterator(const const_iterator& other) {
        ptr = other.ptr;
    }

    const_iterator& operator=(const const_iterator& other) {
        ptr = other.ptr;
        return *this;
    }

    ~const_iterator() {

    }
```

Supporto al const-iterator di tipo forward. È stato implementato un iteratore che itera sui nodi del grafo, più tecnicamente nel nostro caso itera sull'array parallelo contenente i nodi, e li ritorna.

Il metodo **begin()** ritorna l'inizio dell'array, il metodo end() assegna la fine.

```cpp
// Ritorna l'iteratore all'inizio della sequenza dati
const_iterator begin() const {
    return const_iterator(_idArray);
}

// Ritorna l'iteratore alla fine della sequenza dati
const_iterator end() const {
    return const_iterator(_idArray + _aSize);
}
```

PROBJOT SINGH 869434
P.singh3@campus.unimib.it

```
template<class I>
void add_nodes(I start, I end) {
    for (; start != end; start++) {
        try
        {
        add_node(*start);
        }
        catch (const std::exception&){
            std::cout << "FAILED!" << std::endl;
        }

    }
    #ifndef NDEBUG
        std::cout << "graph::add_nodes(const_interator, const_interator)" << std::endl;
    #endif
}
```

Metodo **add_nodes()** che prende in input due iteratori di tipo qualsiasi, che indicano rispettivamente l'inizio e la fine della sequenza, il quale permette di aggiungere una sequenza di nodi al grafo.