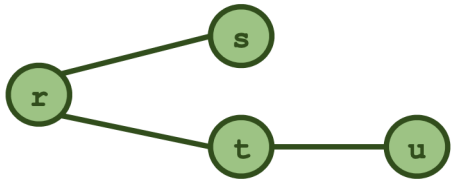
Grafi

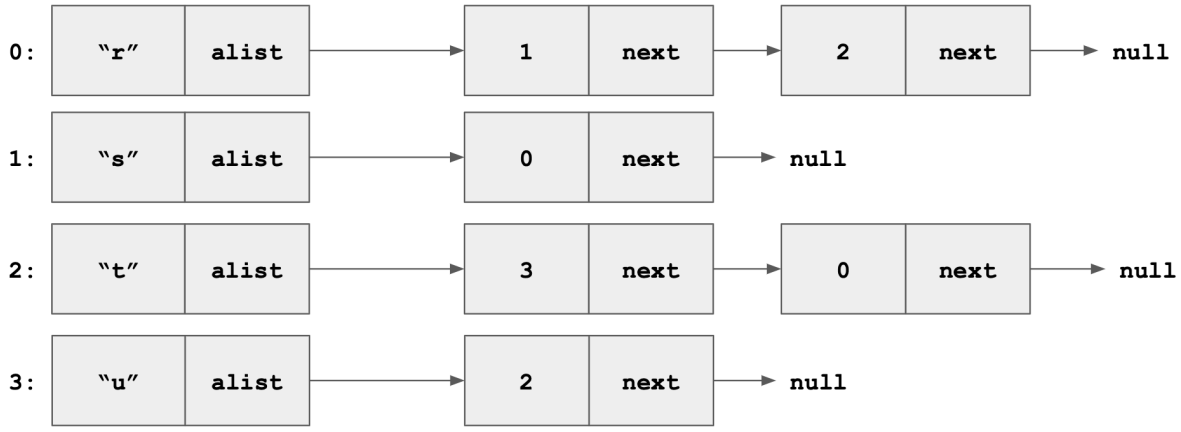
Un grafo orientato è una relazione $E : V \to V$ su un insieme finito V
*Gli elementi di V vengono detti **nodi** e gli elementi di E vengono detti archi*



Graficamente possono essere rappresentati in questo modo ma in termini pratici, ovvero di codice li rappresentiamo con: **liste di adiacenza** o **matrici di adiacenza**

▼ Liste di adiacenza

Le rappresentazione con liste di adiacenza di un grafo orientato $G = (V, E)$ è costituita da un array A di $n = |V|$ insiemi in cui l'elemento i-esimo è il vicinato in uscita del
nodo i $\in$ V cioè $A[i] = N(i)$.



In typescript possono essere create tramite classi e con array, infatti creiamo le classi:

- `ALNode` → nodo lista di adiacenza
- `GNode` → nodo grafo
- `Graph` → grafo

ALNode

```
// Nodo di lista di adiacenza (ADIACENT LIST NODE = ALNode)
class ALNode {
```

```
  index: number
  next: ALNode | null

  constructor(index: number) {
    this.index = index
    this.next = null
  }
}

// index: indice del nodo nella matrice di adiacenza a cui il nodo è associato
// next: contiene il riferimento al prossimo nodo nella lista di adiacenza
```

GNode

```
// Nodo del grafo
class GNode<T> {
  content: T
  alist: ALNode | null

  // per BFS
  d: number
  p: GNode<T>|null

  constructor(content: T) {
    this.content = content
    this.alist = null

    // per BFS
    this.d = Infinity
    this.p = null
  }
}

// content: il contenuto del nodo, può essere qualsiasi tipo di dato generico.
// alist: un puntatore al primo elemento della lista di adiacenza associata
//        al nodo, inizializzato a null
// d: un valore utilizzato nell'algoritmo di BFS (Breadth-First Search)
// p: un puntatore al nodo genitore utilizzato nell'algoritmo di BFS
```

Eccezioni

```
// Eccezioni
class GraphError extends Error { ; }
class InvalidIndexError extends GraphError { ; }
class ExistingArcError extends GraphError { ; }
```

Graph

```
// Grafo
class Graph<T> {
  nodes: GNode<T>[] // array di nodi

  constructor(c: T) {
    let n: GNode<T> = new GNode(c)
    this.nodes = [ n ]
  }

  // Aggiunge un nuovo nodo al grafo (e lo restituisce)
  public addNode(c: T): GNode<T> {
    let n: GNode<T> = new GNode(c)
    this.nodes.push(n)
    return n
```

```
  }

  // Aggiunge un arco orientato (da indice nodo 1 a indice nodo 2)
  public addOrientedEdge(in1: number, in2: number) {

    // controlla se gli indici sono validi
    if (in1 >= this.nodes.length || in1 < 0)
      throw new InvalidIndexError("Arco non creabile: indice nodo 1 non valido")
    if (in2 >= this.nodes.length || in2 < 0)
      throw new InvalidIndexError("Arco non creabile: indice nodo 2 non valido")

    // aggiunge l'arco, evitando self-loop
    if (in1 != in2) {
      // controllo se l'arco c'è già
      let tmp: ALNode|null = this.nodes[in1].alist
      while (tmp != null) {
        if (tmp.index == in2)
          throw new ExistingArcError("Arco già inserito")
        tmp = tmp.next
      }

      // altrimenti inserisco l'arco
      let aln: ALNode = new ALNode(in2)
      aln.next = this.nodes[in1].alist
      this.nodes[in1].alist = aln
    }
  }

  // Aggiunge arco non orientato (ovvero che va in entrambe le direzioni)
  public addUnorientedEdge(in1: number, in2: number) {
    try {
      this.addOrientedEdge(in1,in2)
    } catch(e) {
      if (e instanceof InvalidIndexError)
        throw e
    }
    try {
      this.addOrientedEdge(in2,in1)
    } catch(e) {
      if (e instanceof InvalidIndexError)
        throw e
    }
  }

  // Restituisce una rappresentazione del grafo (liste di adiacenza)
  toString(): string {
    let s: string = "";
    for (let i = 0; i < this.nodes.length; i++) {
      let n = this.nodes[i]
      s += n.content
      let aln: ALNode|null = n.alist
      while(aln != null) {
        s += " -> " + this.nodes[aln.index].content
        aln = aln.next
      }
      s += "\n"
    }
    return s;
  }

// ALGORITMO BFS

}
```

Breadth First Search - BFS (dentro la classe *graph*)

```
// BFS per verificare se un valore c è raggiungibile a partire da s
  public reachable(s: number, c: T): boolean {
    // inizializza vertici
    for (let i=0; i<this.nodes.length; i++) {
      this.nodes[i].d = Infinity
      this.nodes[i].p = null
    }
```

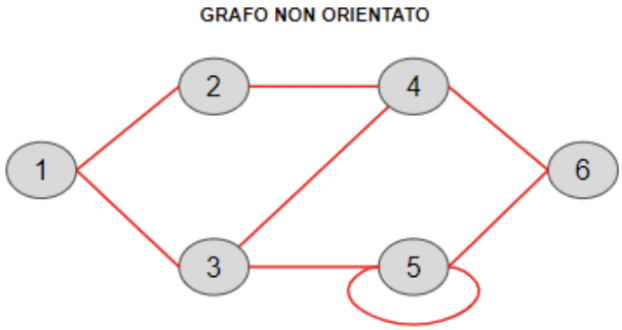
```
    this.nodes[s].d = 0
    // inizializza la coda
    let Q: GNode<T>[] = [ this.nodes[s] ]
    // finché la coda non è vuota
    while (Q.length != 0) {
      // estrae un nodo
      let u: GNode<T> = Q.shift()! // evito che evidenzi errore, sapendo che Q non è vuota (https://www.typescriptlang.org/docs/handbook
      // se u contiene c, termina restituendo true
      if (u.content == c)
        return true
      // altrimenti, continua aggiungendo la lista di adiacenza di u a Q (se necessario)
      let alu: ALNode|null = u.alist
      while (alu != null) {
        let v: GNode<T> = this.nodes[alu.index]
        // se v è BIANCO (d e p avranno i valori iniziali)
        if (v.p == null) {
          // v diventa GRIGIO (cambiano i valori di d e p)
          v.d = u.d + 1
          v.p = u
          // e viene messo in Q
          Q.push(v)
        }
        alu = alu.next
      }
    }
    // se non l'ha trovato, restituisce false
    return false
  }
```

▼ Matrici di adiacenza

è una matrice quadrata A con n righe e n colonne, numerate da 0 a $n-1$, dove l'elemento $A_{i,j}$ (in riga i e colonna j) assume un valore in {0, 1} con il seguente significato:

📍 **Appartenenza di un arco**

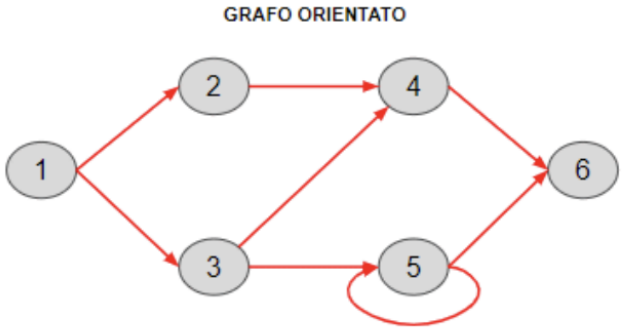
$$A_{i,j} = \begin{cases} 1, & se \quad l'arco \ (i,j) \in E \\ 0 & se \quad l'arco \ (i,j) \notin E \end{cases}$$



nodi destinazione

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

E' una matrice trasposta



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

In typescript può essere creata tramite classi e array multidimensionali, infatti creiamo prima una classe **Matrix** per rappresentare matrici con

- **r** righe
- **c** colonne

```
// Classe per rappresentare matrici di r righe e c colonne
class Matrix {
  r: number;
  c: number;
  A: number[][];

  constructor(r = 1, c = 1) {
    this.r = r;
    this.c = c;
    this.A = [];

// Creazione di un array multidimensionale con r righe e c colonne
// con valore iniziale 0
    for (let i = 0; i < r; i++) {
      this.A[i] = [];
      for (let j = 0; j < c; j++) {
        this.A[i][j] = 0;
      }
    }
  }

  // Imposta il valore contenuto nella cella i,j
  setValue(i: number, j: number, v: number): void {
    this.A[i][j] = v;
  }

  // Restituisce il valore contenuto nella cella i,j
```

```
  getValue(i: number, j: number): number {
    return this.A[i][j];
  }

  // Verifica se la matrice è quadrata
  quadrata(): boolean {
    return this.r == this.c;
  }

  // Azzera tutti gli elementi della matrice
  setZero(): void {
    this.A.forEach(riga => riga.map(e => 0));
  }

  // Genera una matrice identità, ovvero una matrice con
  // 1 sulla diagonale principale
  // 0 nelle altre celle
  setId(): void {
    if (this.quadrata()) {
      this.setZero();
      for (let i = 0; i < this.r; i++)
        this.A[i][i] = 1;
    }
  }

  // Imposta il contenuto delle celle nella riga i
  // copiando i valori dalla "riga" passata
  setRiga(i: number, riga: number[]) {
    if (0 <= i && i < this.r && riga.length == this.c)
      for (let j = 0; j < this.c; j++)
        this.A[i][j] = riga[j]
  }

  // Imposta il contenuto delle celle nella colonna j
  // copiando i valori dalla "colonna" passata
  setColonna(j: number, colonna: number[]) {
    if (0 <= j && j < this.c && colonna.length == this.r) {
      for (let i = 0; i < this.r; i++)
        this.A[i][j] = colonna[i];
    }
  }

  // Imposta i valori contenuti nelle celle della matrice
  // copiandoli valori dalla matrice (array di array) passata
  setAll(dati: number[][]) {
    if (this.r == dati.length)
      for (let i = 0; i < this.r; i++)
        this.setRiga(i, dati[i]);
  }

  // Restituisce la matrice ottenuta moltiplicando la matrice
  // corrente con la matrice B passata come argomento
  mul(B: Matrix): Matrix {
    let A = this;
    // Mmoltiplicazione fatta solo se si può fare
    if (A.c == B.r) {
      let R = new Matrix(A.r, B.c);
      for (let i = 0; i < R.r; i++) {
        for (let j = 0; j < R.c; j++) {
          let t = 0;
          for (let h = 0; h < A.c; h++)
            t += (A.getValue(i, h) * B.getValue(h, j));
          R.setValue(i, j, t);
        }
      }
      return R;
    }
    // Altrimenti, restituisce un'eccezione
    throw new Error("Non posso moltiplicare!");
  }

  // Restituisce una stringa che rappresenta la matrice
  toString() {
    return `${this.r}x${this.c}:\n|${this.A.join("|\n|")}|`
  }
}
```

Spiegazione classe matrix

```
for (let i = 0; i < r; i++) {
  this.A[i] = [];
  for (let j = 0; j < c; j++) {
    this.A[i][j] = 0;
  }
}

// PRIMO CICLO FOR (righe)
Il primo ciclo for itera attraverso le righe della matrice (r volte) e,
per ogni riga, crea un array vuoto e lo inserisce all'interno dell'array A.
In questo modo, ogni elemento dell'array A corrisponde ad una riga della matrice.

// SECONDO CICLO FOR (COLONNE)
Il secondo ciclo for itera attraverso le colonne della matrice (c volte) e,
per ogni cella, imposta il valore della cella a 0. In questo modo, ogni elemento
dell'array A rappresenta una riga della matrice, e ogni riga contiene c colonne,
ciascuna con valore iniziale pari a 0.

Quindi, alla fine di questi due cicli for, l'array A rappresenta una matrice
vuota di r righe e c colonne, dove ogni cella è inizializzata a 0.

_____________________________________________________________________________

setRiga(i: number, riga: number[]) {
    if (0 <= i && i < this.r && riga.length == this.c)
      for (let j = 0; j < this.c; j++)
        this.A[i][j] = riga[j]
  }

// CONTROLLO DELLA RIGA
La funzione setRiga(i, riga) prende in input un indice di riga i e un array
riga contenente i valori da impostare nella riga i della matrice.
La funzione controlla innanzitutto che i sia un indice di riga valido per
la matrice corrente (ovvero, che 0 <= i < this.r). Successivamente, controlla
che l'array riga abbia la stessa lunghezza della matrice corrente in direzione
delle colonne (ovvero, che riga.length == this.c).

// CICLO FOR
Se entrambi i controlli sono verificati, la funzione scorre ogni colonna
della riga i e imposta il valore della cella corrispondente a quello
dell'elemento j dell'array riga. In altre parole, la funzione sostituisce
i valori presenti nella riga i della matrice con quelli contenuti nell'array riga.
```

Grazie a **Matrix**, possiamo rappresentare il grafo con

- `Node<T>` → nodo grafo
- `Graph<T> extends Matrix` → grafo

Nodo grafo

```
// Classe che rappresenta un nodo del grafo
class Nodo<T> {
  id: number; // indice del nodo nella matrice di adiacenza
  value: T|null; // valore del nodo

  constructor() {
    this.id = -1;
    this.value = null;
  }

  // this.id -1 come valore iniziale per indicare che non è stato ancora inserito
```

```
  // this.value rappresenta il valore associato al nodo.
  // Inizialmente, il valore di value è null perché il nodo
  // non ha ancora un valore assegnato.
}
```

Grafo

```
// Classe che rappresenta il grafo come matrice
// NB: Estende Matrix, creando una matrice che è sempre quadrata
// NB: matrice quadrata: numero di righe e numero di colonne uguali
class Graph<T> extends Matrix {

  constructor(v: T) {
    super(1,1);
    let n = new Nodo<T>();
    n.id = this.r;
    n.value = v;
  }

  // Restituisce il numero di nodi
  getNNodi(): number {
    return this.r;
  }

  // Restituisce il numero di archi (usando reduce per sommare gli 1
  // presenti nelle righe e nelle colonne)
  getNArchi(): number { //
    return this.A.map(r => r.reduce((a,b) => a + b)).reduce((a,b) => a+b);
  }

  // Aggiunge un nuovo arco orientato da idn1 a idn2
  // (settando a 1 la cella corrispondente)
  addOrientedEdge(idn1: number, idn2: number): void {
    if (idn1 >= this.r || idn2 >= this.r || idn1 < 0 || idn2 < 0)
      throw new Error("Arco non creabile");
    if (idn1 != idn2)
      super.setValue(idn1,idn2,1);
  }

  // Aggiunge un nuovo arco non-orientrato tra idn1 e idn2
  // (settando a 1 le celle corrispondenti)
  addUnorientedEdge(idn1: number, idn2: number) {
    this.addOrientedEdge(idn1, idn2);
    this.addOrientedEdge(idn2, idn1);
  }

  // Aggiunge un nuovo nodo, inserendo un nuovo indice
  // nella matrice di adiacenza (NB: sia su righe sia su colonne)
  addNode(v: T): Nodo<T> {
    let n = new Nodo<T>();
    n.id = this.r;
    n.value = v;
    let rc: number[] = [];
    for (let i = 0; i < this.r; i++) {
      this.A[i].push(0);
      rc.push(0);
    }
    rc.push(0);
    this.A.push(rc);
    this.r++;
    this.c++;

    return n;
  }

  // Rimuove un nodo dal grafo, rimuovendo l'indice corrispondente
  // dalla matrice di adiacenza (NB: sia su righe sia su colonne)
  removeNode(idn: number) {
    if (idn > this.r)
      return false;

    while (idn < this.r - 1) {
      // shifting righe a sx
```

```
      for (let i = 0; i < this.r; i++) {
        this.A[i][idn] = this.A[i][idn + 1];
      }
      // shifting colonne su
      for (let i = 0; i < this.r; i++) {
        this.A[idn][i] = this.A[idn + 1][i];
      }
      idn++;
    }

    // elimina ultima riga e ultima colonna
    this.A.length--;
    this.r--;
    for (let i = 0; i < this.r; i++)
      this.A[i].length--;
    this.c--;
    return true;
  }

  // Genera archi casuali per i nodi presenti
  randomArcs() {
    for (let i = 0; i < this.r; i++) {
      this.A[i] = [];
      for (let j = 0; j < this.c; j++) {
        if (i==j)
          this.A[i][j] = 0;
        else
          this.A[i][j] = Math.round(Math.random());
      }
    }
  }
}
```