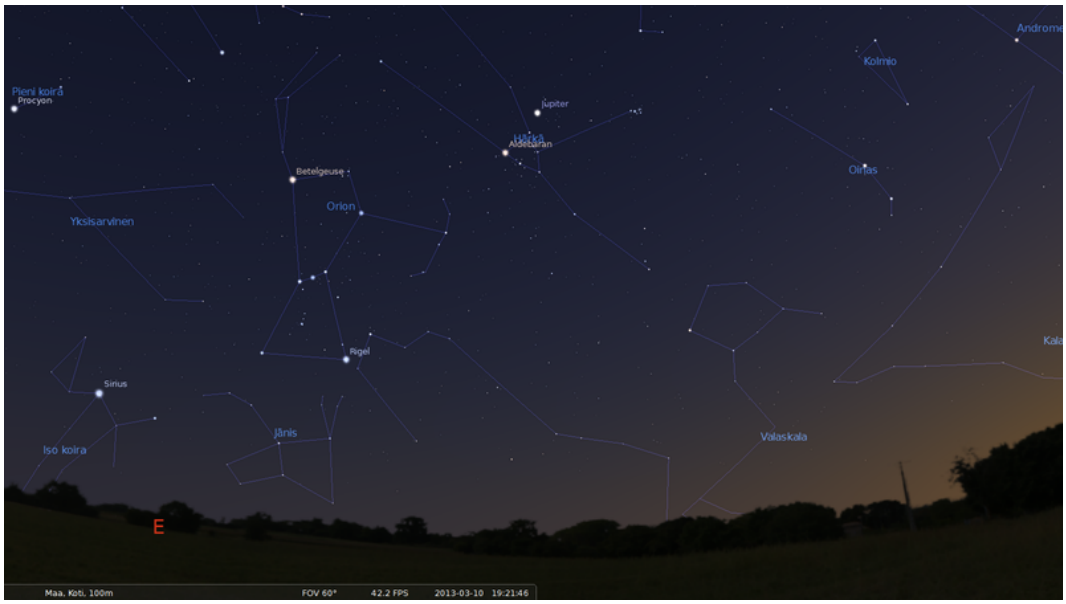
I Grafi



Giovanni Scavello

Indice

Introduzione

Nel presente lavoro si intende elaborare una unità didattica sui grafi. Lo scopo principale dell'unità didattica è quello di fornire agli studenti uno strumento concettuale attraverso l'esplorazione di situazioni provenienti dalla loro esperienza quotidiana. Il concetto di grafo non rappresenta soltanto una struttura dati fondamentale dell'informatica ma è soprattutto un modello concettuale attraverso cui si possono analizzare molti problemi non solo nell'ambito strettamente algoritmico[1].

Lo sviluppo di un'attività didattica deve procedere con un obiettivo ben preciso: creare nei ragazzi la curiosità nei confronti dell'argomento, far sorgere in loro dubbi a cui cercare risposte. Qualunque attività didattica che perda di vista questo obiettivo diventa un mero elenco di concetti e nozioni che altri prima di noi hanno, con molta più competenza e rigore, sviluppato nel lungo cammino della conoscenza umana. Dobbiamo ricordare a noi stessi che «siamo come nani sulle spalle dei giganti» e dovremmo aiutare i ragazzi che abbiamo di fronte a salire sulle spalle di questi giganti. Bisognerebbe far capire a i ragazzi che i temi affrontati in classe non sono banali ma allo stesso tempo che questi temi hanno una certa vitalità e non devono essere trattati come concetti usa e getta utili per ottenere un voto. Se si riesce a creare un'empatia nei confronti dei concetti, questi entrano a far parte della quotidianità del ragazzo. Si generano così dei nuovi punti di partenza per ulteriori riflessioni dando vita ad un vero e proprio processo di ampliamento degli orizzonti culturali. Per raggiungere questo obiettivo è necessario uno sforzo da parte del docente, il quale deve prima di tutto depurare le sue lezioni da tutti quegli elementi che possono rivelarsi noiosi e inutili ai fini di catturare l'attenzione dei ragazzi. Diceva Saint-Exupery che «la perfezione è raggiunta non quando non c'è più niente da aggiungere, bensì quando non c'è più nulla da togliere» ed è secondo questo criterio che si è cercato di sviluppare la presente unità didattica.

Gli argomenti vengono presentati partendo da casi concreti e cercando di estrapolare da essi concetti più generali. Con questa scelta si vuole stimolare i ragazzi ad analizzare le situazioni particolari e procedere verso la creazione di modelli astratti. E soprattutto, presentando problemi e situazioni reali diversi tra loro ma con lo stesso sfondo teorico, diventa più semplice riuscire ad intercettare i diversi interessi dei ragazzi.

L'unità didattica si rivolge ad una classe quinta di un istituto tecnico industriale o, in generale, ad una classe quinta di una scuola in cui le basi di

1Si veda come esempio http://www.nytimes.com/interactive/2011/10/23/sunday-review/an-overview-of-the-euro-crisis.html?_r=0

programmazione siano abbastanza solide e la conoscenza delle strutture dati sia già stata affrontata in precedenza. Tuttavia, i concetti presentati possono essere trattati anche in altri ordini di scuole, magari sacrificando la formalizzazione e la programmazione ma concentrandosi di più sugli aspetti concettuali e grafici.

Come prerequisiti si richiede dimestichezza con la programmazione in C/C++, conoscenza dei puntatori, conoscenza delle strutture dati elementari (liste, pile, code).

La struttura dell'unità didattica è strettamente sequenziale: ogni sezione è basata sulla precedente e di conseguenza è richiesta la comprensione della sezione precedente per poter affrontare quella successiva. In realtà ciò è valido soprattutto per quanto riguarda la rappresentazione dei grafi, argomento che costituisce la base concettuale per l'applicazione di tutti gli algoritmi che saranno presentati nello svolgersi dell'argomento.

Il tempi previsto per lo svolgimento dell'unità didattica è di circa 12 ore che equivalgono a due settimane in un istituto tecnico o circa due mesi in un liceo scientifico tecnologico.

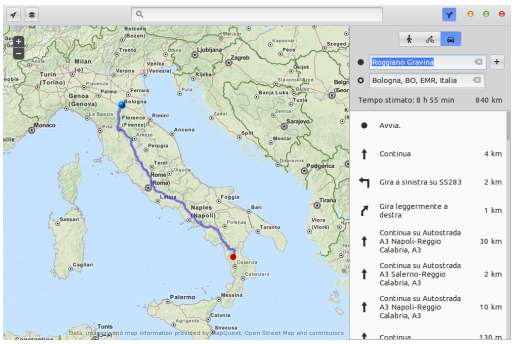
Gli obiettivi minimi possono consistere nella capacità di generare una descrizione del grafo in base alle caratteristiche dello stesso presentate graficamente ed implementare le modalità di visita presentate durante l'unità didattica. Entrambi gli obiettivi rappresentano, a mio giudizio il punto chiave della comprensione dell'argomento.

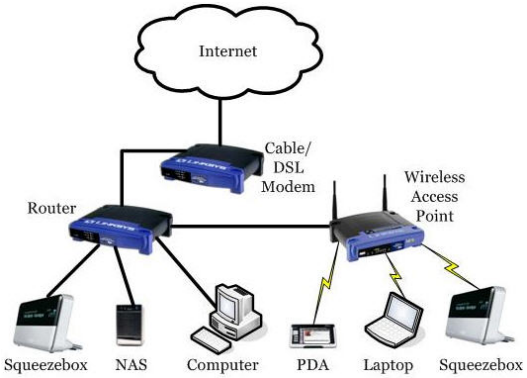
Ogni capitolo è corredato da una serie di esercizi che mirano alla comprensione degli argomenti. Nel capitolo 4 vengono proposti esercizi di programmazione in cui viene chiesto allo studente uno sforzo aggiuntivo per modificare gli algoritmi di visita dei grafi.

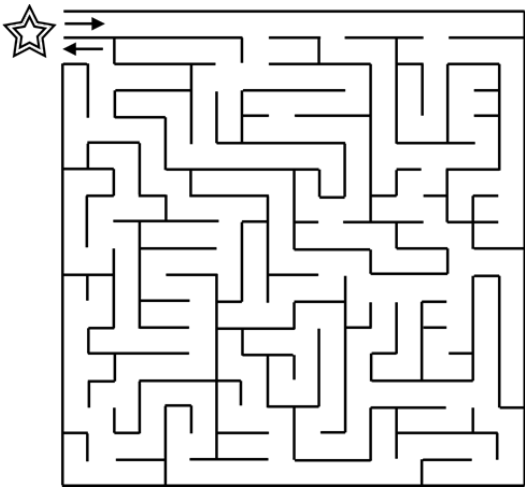
1 Problemi reali e loro rappresentazione

Tempo previsto: 1 ora di lezione frontale

Cosa hanno in comune le cartine stradali, Internet e i labirinti? Guardiamoli attentamente:







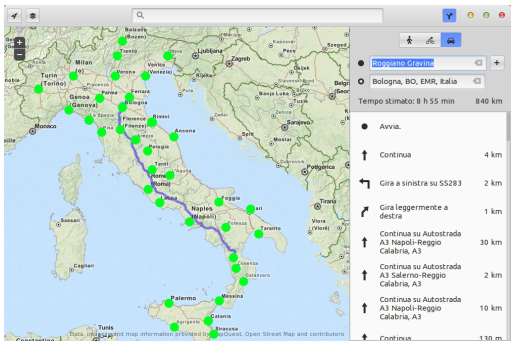
Apparentemente nulla. Eppure

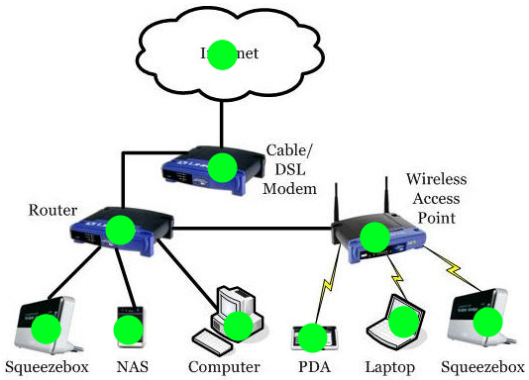
- Le cartine stradali ci permettono di stabilire qual è la strada da percorrere per andare da una città ad un'altra. Decidiamo da dove partire, seguiamo un percorso tra varie tappe intermedie ed arriviamo a destinazione.
- Internet ci permette, tramite WWW, di navigare da un documento ad un altro semplicemente facendo click su un link. Quando ci colleghiamo ad una pagina web qualsiasi l'informazione viaggia attraverso i cavi e le connessioni telefoniche fino a raggiungere un computer che è fisicamente situato in un'altra parte del mondo.
- Il labirinto è un esercizio di orientamento in cui bisogna trovare l'unico cammino che ci permette di arrivare all'uscita. Il punto di partenza questa

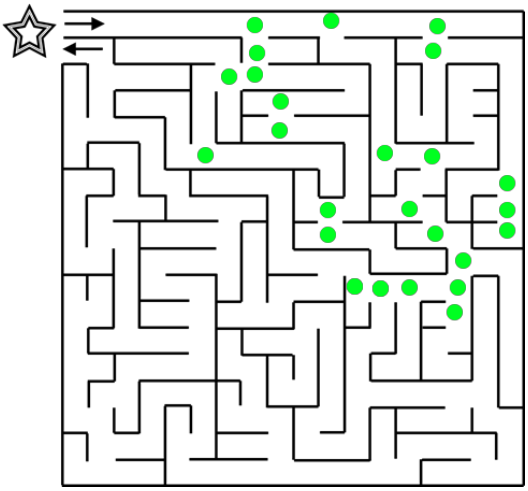
volta lo conosciamo, è unico e coincide con il punto di arrivo. Ma come facciamo a trovare l'unico cammino che ci conduce all'uscita?

Notiamo che sono state sottolineate alcune parole che hanno significati analoghi e che richiamano concetti che hanno qualcosa in comune: partenza, arrivo, cammino, percorso, collegamento.

Proviamo adesso a modificare le immagini, utilizzando dei puntini colorati al posto degli oggetti reali:







Le immagini iniziano ad avere alcune caratteristiche comuni. In altre parole, in ogni immagine siamo riusciti ad "isolare" un *oggetto di interesse*:

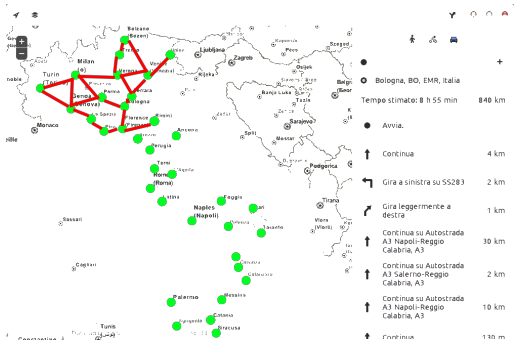
- Nella cartina, sono evidenziate le città (alcune)
- Su internet, sono evidenziati i dispositivi (alcuni)
- Nel labirinto sono evidenziati gli incroci (alcuni)

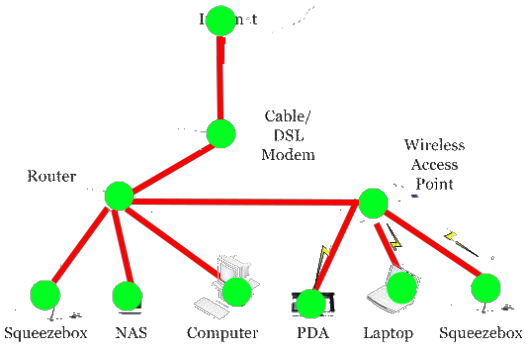
In tutti e tre i casi questi "oggetti" hanno una qualche relazione:

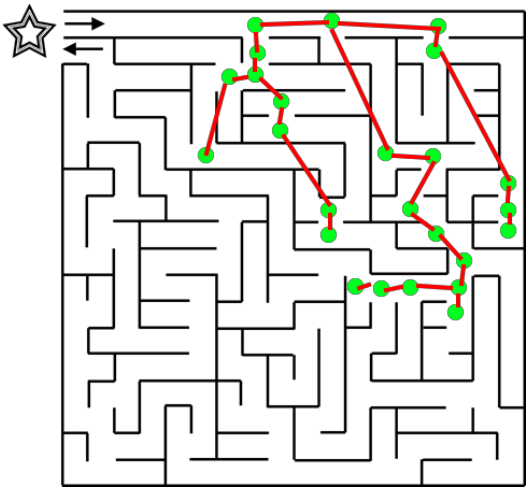
- Sulla cartina, sono le strade fra una città e l'altra
- Su internet, sono i collegamenti fisici fra i dispositivi
- Sul labirinto, sono i punti raggiungibili a seconda della scelta (sinistra,

destra, sopra, sotto)

Se eliminiamo l'informazione sul contesto in cui ci troviamo, cioè se lasciamo i puntini ed evidenziamo le relazioni fra gli oggetti, otteniamo una struttura comune.







In tutti e tre i casi abbiamo ottenuto uno “schema” composto da due “elementi”:

- Un insieme di oggetti rappresentativi dello specifico problema
- Un insieme di relazioni (fisiche o logiche) fra questi oggetti

In altre parole, abbiamo fatto emergere una struttura comune che permette di descrivere i tre problemi mediante lo stesso modello. Ciò significa che siamo riusciti a trovare una descrizione dei problemi che prescinde dalla natura concreta degli stessi. La struttura astratta di cui stiamo parlando è il *grafo*.

Per quanto semplice ed intuitivo, il concetto di grafo gioca un ruolo fondamentale in un'ampia varietà di applicazioni computazionali. Un volta definito questo oggetto astratto, è possibile utilizzarlo per aumentare la conoscenza rispetto al fenomeno che si sta studiando. Ad esempio è possibile

1. indagare se esiste una relazione indiretta fra due elementi del grafo
2. sapere quali sono gli elementi del grafo che hanno una interdipendenza

reciproca

3. calcolare quali sono i possibili punti di arrivo dato un elemento iniziale
4. calcolare qual è il minimo numero di connessioni fra due elementi

Nonostante la loro semplicità i grafi, e gli algoritmi che su essi possono essere implementati, rappresentano la base per un numero sterminato di applicazioni computazionali.

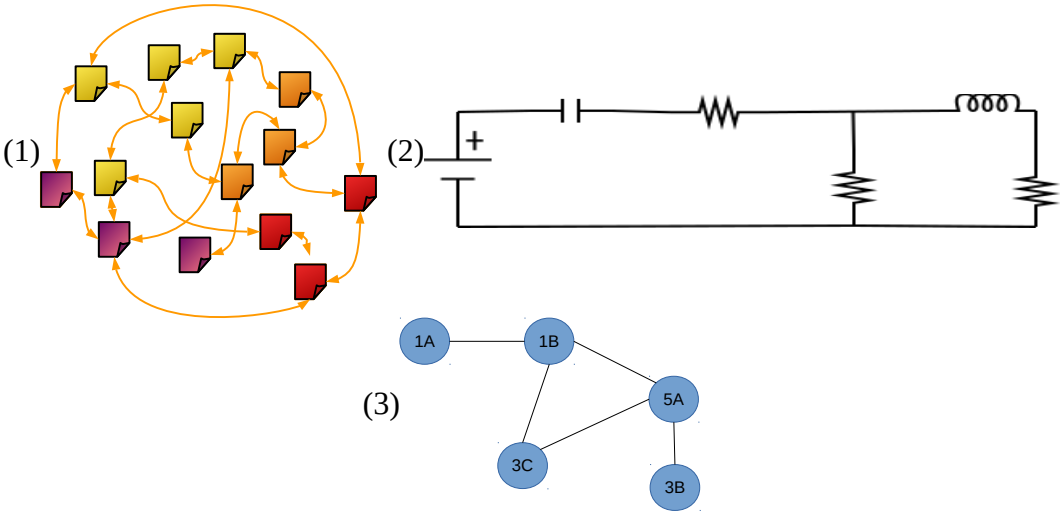
1.1 Altri Esempi

Di seguito vengono riportati altri esempi di situazioni reali che possono essere modellate tramite un grafo.

Quando navighiamo sul WWW ci spostiamo fra una pagina e l'altra mediante collegamenti logici (link ipertestuali). L'intero Web può essere considerato un grafo con miliardi di nodi e collegamenti. Gli algoritmi di ricerca sui grafi sono una componente fondamentale dei motori di ricerca e la loro implementazione efficiente è alla base dell'usabilità del motore stesso (figura 1).

Un circuito elettrico può essere espresso mediante un grafo in cui i nodi sono gli elementi elettrici. Gli strumenti di analisi automatica dei circuiti testano le connessioni fra i nodi per individuare cammini critici, come ad esempio quelli che danno luogo a cortocircuiti oppure per individuare la topologia di connessione che permetta di disporre e collegare gli elementi senza che i fili si sovrappongono (figura 2).

La segreteria di una scuola deve pianificare i consigli di classe ma ha il vincolo che alcuni docenti appartengono a più consigli di classe e quindi non possono partecipare ai consigli che si svolgono in contemporanea. La teoria dei grafi ci aiuta a formulare questo problema in termini matematici ed a trovare una soluzione algoritmica a questo problema (figura 3).



2 Dalla pratica alla teoria

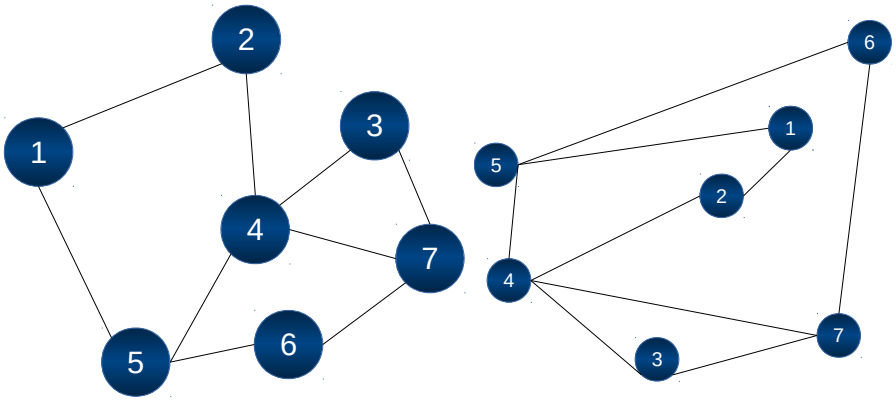
Tempo previsto: 1 ora di lezione frontale

2.1 Definizione informale

Un grafo è una collezione di nodi ed archi. Possiamo cioè definire un insieme di oggetti a cui associare un secondo insieme che definisce le relazioni fra questi oggetti. Graficamente

- i nodi sono rappresentati da punti a cui sono associate delle etichette (numeri, nomi, etc…)
- gli archi sono rappresentati da linee che connettono due nodi

Un arco,quindi, non è altro che una coppia di nodi fra cui esiste una relazione.



I due grafi riportati in figura sono entrambi descritti dai seguenti insiemi:

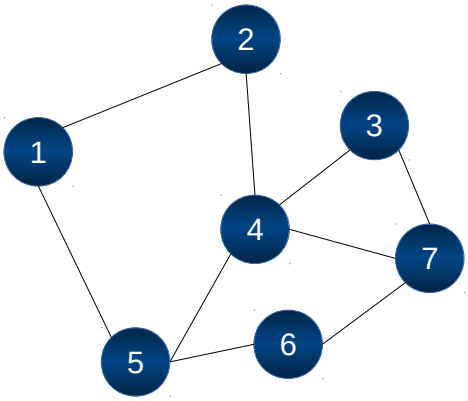
Nodi = {1,2,3,4,5,6,7}

Archi ={1-2, 1-5, 2-4, 3-4, 3-7, 4-7, 4-5, 5-6, 6-7}

È importante far notare che un grafo è un oggetto *astratto* e che gli stessi insiemi possono essere disegnati in maniera molto diversa fra loro pur rappresentando lo stesso oggetto. La disposizione grafica dei nodi e degli archi di un grafo è indipendente dalla rappresentazione del grafo. Ciò significa che lo stesso grafo può essere "disegnato" in più modi a partire dalla stessa rappresentazione.

2.2 Nomenclatura

Per poter lavorare con i grafi è necessario fissare alcuni concetti di base. Nel seguito si farà riferimento al grafo indicato in figura per mostrare operativamente tali concetti. La scelta di lavorare inizialmente con elementi visivi ha una duplice spiegazione: da un lato si ha la possibilità di esprimere i concetti in maniera intuitiva, dall'altro si mostra che esiste una rappresentazione "numerica" delle informazioni associate ai grafi.



Chiamiamo *cammino* una lista di vertici in cui vertici successivi sono connessi da archi del grafo

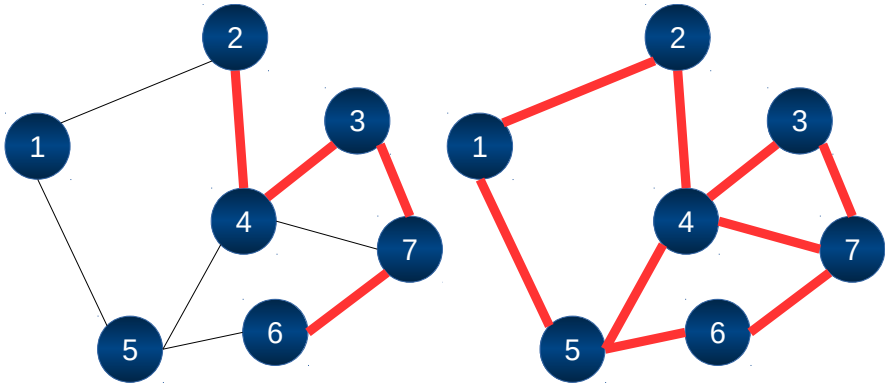
ESEMPIO

Cammino = {2,4,3,7}

Diciamo che un insieme di nodi è *connesso* se esiste un cammino da ogni nodo dell'insieme ad ogni altro nodo dell'insieme

ESEMPIO

CC = {1, 2, 3, 4, 5, 6, 7}



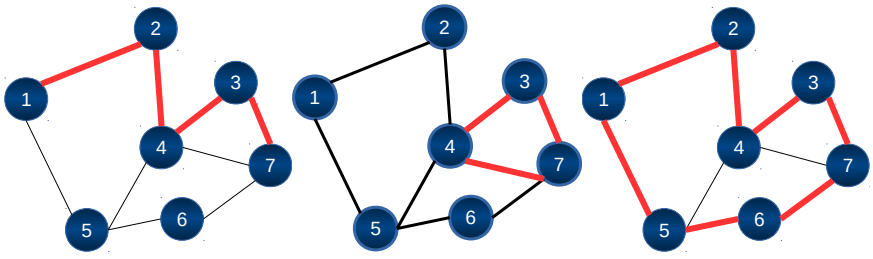
Un cammino è detto *semplice* se non ci sono nodi che si ripetono nella lista.

Un cammino è detto *ciclo* se il primo nodo della lista è uguale all'ultimo.

ESEMPIO

Ciclo1 = {1,2,4,3,7,6,5,1}

Ciclo2 ={3,4,7,3}



In molti problemi reali, non è sufficiente stabilire se esiste o meno una

relazione fra gli oggetti ma è necessario assegnare un verso a tale relazione.

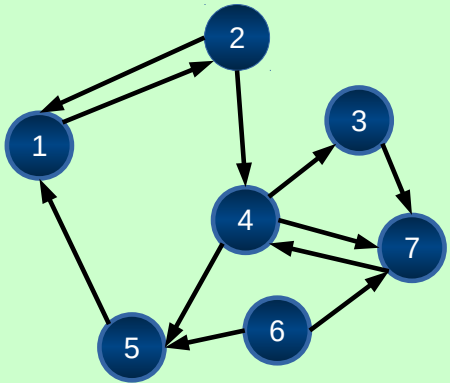
Estendendo la definizione data nel paragrafo 2.1, possiamo chiamare *orientato* un grafo in cui gli archi non solo rappresentano l'esistenza di una qualche relazione ma stabiliscono l'ordine di tale relazione. In questo tipo di grafi l'esistenza di un arco tra un nodo A ed un nodo B non implica che esista un arco tra il nodo B ed il nodo A.

ESEMPIO

L'esempio più immediato di una tale situazione è quello di una mappa in cui siano presenti sia delle strade a doppio senso di circolazione che delle strade a senso unico. In questo caso l'insieme degli archi è formato da coppie ordinate, ossia da coppie in cui l'ordine dei nodi definisce la direzione dell'arco

Nodi={1, 2, 3, 4, 5, 6, 7}

Archi={1-2, 2-1, 2-4, 4-3, 4-7, 4-5, 3-7, 7-4, 6-7, 6-5, 5-1}

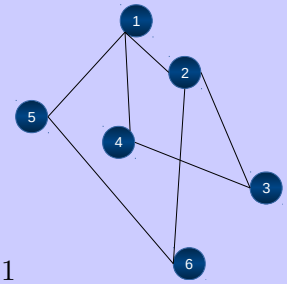


2.3 Esercizi

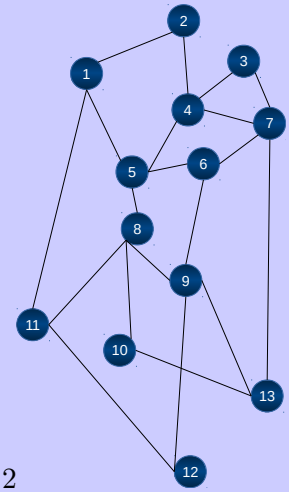
1. Disegnare i seguenti grafi:
   Nodi = {1,2,3,4,5,6,7}
   Archi ={1-2, 1-3, 2-4, 3-4, 2-5, 6-5, , 6-7}

   Nodi = {1,2,3,4,5,6,7, 8, 9, 10, 11}
   Archi ={1-3, 2-4, 4-3, 3-5, 6-5, 6-7, 8-7, 11-12, 10-11, 6-12, 9-10, 8-9}

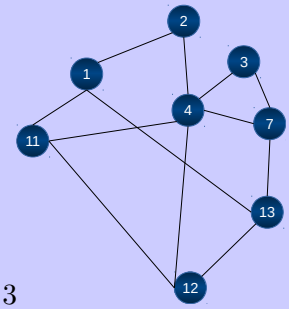
2. Per ognuno dei grafi indicati in figura, indicare l'insieme dei nodi e degli archi



1

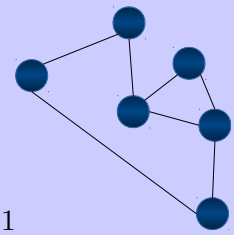


2

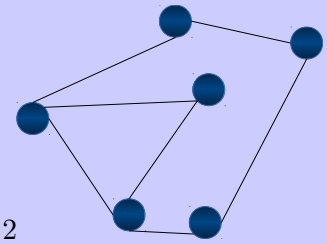


3

3. Dati i due grafi rappresentati in figura, assegnare le etichette ai nodi in modo che abbiano gli stessi insiemi di nodi e archi.



1



2

3 Rappresentazione

Tempo previsto: 1 ora di lezione frontale + 2 ore di laboratorio

Nel precedente capitolo, si è parlato di grafi senza fare alcun cenno all'implementazione su un calcolatore. Utilizzare una rappresentazione visiva dei grafi permette di coglierne intuitivamente i vantaggi come modello astratto. Il disegno di un grafo è facilmente interpretabile e, almeno per grafi di ridotte dimensioni, permette di poter applicare immediatamente concetti come cammino, ciclo, connettività. Tuttavia, la grande importanza dei grafi consiste nel poter automatizzare queste operazioni in modo da poter trattare problemi che graficamente sarebbero impossibili da maneggiare. Si pensi ad esempio calcolo del percorso di un itinerario attraverso molte città o ai suggerimenti di connessione che compaiono sui *social network*.

Per poter implementare un qualsiasi algoritmo che lavori su un grafo bisogna innanzitutto stabilire in che modo il grafo deve essere rappresentato nella memoria di un calcolatore. Si è visto in precedenza che un grafo è un insieme di nodi a cui è associato un insieme di archi. Questi ultimi non sono altro che un modo per rappresentare relazioni fra una coppia di nodi. Abbiamo visto che un grafo può essere facilmente descritto da due insiemi ma questa descrizione non è molto utile dal punto di vista computazionale. Il primo passo per l'implementazione di algoritmi su grafi consiste quindi nello scegliere una struttura dati opportuna per mezzo della quale memorizzare la struttura del grafo indipendentemente dalla rappresentazione grafica. Seppure in apparenza banale, la scelta della rappresentazione è cruciale e influisce su molti aspetti computazionali tra cui i più importanti sono

1) Lo spazio occupato in memoria
2) I tempi di esecuzione degli algoritmi
3) La possibilità di aggiornare facilmente le informazioni sul grafo

Inoltre, il passaggio dalla rappresentazione visiva a quella “informatica” è un punto fondamentale per imparare ragionare in termini di nodi e connessioni. C'è infatti una differenza enorme fra “vedere” un grafo e lavorare sulla sua rappresentazione. Il cervello umano interpreta il disegno di un grafo nel suo complesso e molti problemi sembrano di facile soluzione. Ma quando le dimensioni del grafo crescono e si perde quindi la visuale “globale”, bisogna passare ad una visione più strutturata che permetta di analizzare il grafo attraverso un approccio “locale” e sistematico.

ESEMPIO

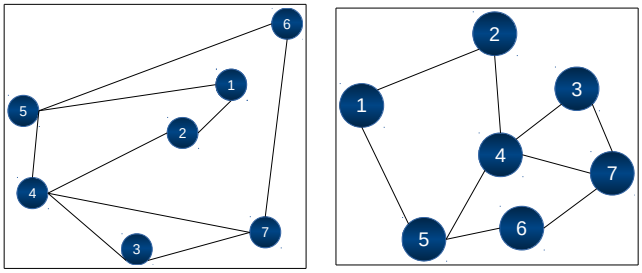
Supponiamo di voler fare un viaggio e che la nostra meta sia una remota città

di una nazione lontana. Non abbiamo a disposizione una mappa che ci indichi il cammino ma possiamo intraprendere il nostro viaggio chiedendo di volta in volta la direzione da seguire per raggiungere il punto di arrivo. Quando ci fermiamo in una città e chiediamo in che direzione andare, gli abitanti del posto non conoscono l'intera rete stradale ma ci danno delle indicazioni basate sulla conoscenza locale del loro territorio, suggerendoci poi di «chiedere più avanti». Questa è esattamente la situazione in cui ci si trova quando si deve lavorare sui grafi: ogni nodo ci dà informazioni locali (sappiamo a quale nodo è connesso) unendo le quali possiamo estrarre informazioni sull'intero grafo.

Nei prossimi paragrafi verranno introdotte e analizzate le due rappresentazioni classiche, la matrice di adiacenza e le liste di adiacenza, che costituiscono la base per un approccio sistematico all'utilizzo dei grafi.

3.1 Matrice di adiacenza

Un modo abbastanza immediato per rappresentare un grafo, consiste nel considerare un matrice G in cui il valore dell'elemento G[i][j] è 1 se nel grafo esiste un arco fra i nodi i e j, 0 altrimenti.



	1	**2**	**3**	**4**	**5**	**6**	**7**
1	0	1	0	0	1	0	0
2	1	0	0	1	0	0	0
3	0	0	0	1	0	0	1
4	0	1	1	0	1	0	1
5	1	0	0	1	0	0	1
6	0	0	0	1	1	0	1
7	0	0	1	1	0	1	0

Notiamo subito che la matrice ha il valore 1 sia in posizione [i][j] che in posizione [j][i]. È presente cioè una informazione duplicata: nei grafi non orientati, l'esistenza di un arco da i a j implica l'esistenza dell'arco da j a i[2]. Questa è una situazione tipica dell'informatica: se memorizzo più informazione, occupo più memoria ma ho accesso diretto all'informazione stessa; se memorizzo meno informazione, occupo meno memoria ma devo "perdere del tempo" per ricostruirla.

Nel caso della matrice di adiacenza, con la rappresentazione che abbiamo visto,

2Nei grafi non orientati la matrice di adiacenza non è in genere simmetrica e l'esistenza di un arco da i a j non implica l'esistenza dell'arco fra j e i.

diventa immediato verificare se esiste un arco da un nodo ad un altro ma bisogna scandire un'intera riga della matrice per sapere quali sono tutti i nodi adiacenti ad un nodo dato.

ESEMPIO

(1)

	1	**2**	**3**	**4**	**5**	**6**	**7**
1	0	1	0	0	1	0	0
2	0	0	0	1	0	0	0
3	0	0	0	1	0	0	1
4	0	0	0	0	1	0	1
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0

(2)

	1	**2**	**3**	**4**	**5**	**6**	**7**
1	0	1	0	0	1	0	0
2	1	0	0	1	0	0	0
3	0	0	0	1	0	0	1
4	0	1	1	0	1	0	1
5	1	0	0	1	0	0	1
6	0	0	0	1	1	0	1
7	0	0	1	1	0	1	0

Nella matrice (1), per verificare l'esistenza di un arco tra i nodi 2 e 1 devo controllare sia l'elemento G[2][1] che l'elemento G[1][2]. Nella matrice (2) posso controllare direttamente l'elemento G[2][1].

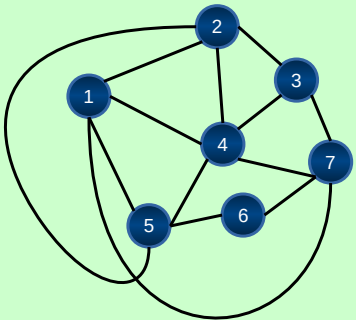
La rappresentazione mediante matrice di adiacenza è molto semplice perché non necessita di strutture dati complesse, essendo implementata mediante un *array* bidimensionale. A questo punto è possibile scrivere una semplice funzione che ci permetta di leggere un grafo da tastiera. La creazione di un grafo mediante matrice di adiacenza può essere implementata in linguaggio C++ .

```
bool** creaGrafoMatrice(int N)
{
	//alloco spazio in memoria
	bool** M = new bool*[N];

	for (int i = 0; i < N; i++)
		M[i] = new bool [N];

	//inizializzo il grafo senza archi
	for (int i = 0; i < N; i++)
		for (int j = 0; j < N; j++)
			M[i][j] = false;
	//l'arco (i,j) viene inserito settando a true gli elementi [i][j] e [j][i]
	while (true)
	{
		int i,j;
		printf("Inserisci arco da i a j (-1 -1 per uscire): ");
		scanf("%i %i", &i, &j);

		if (i<0 || j < 0)
			break;

		M[i][j] = true; M[j][i] = true;
	}

	return M;
}
```

Balza subito agli occhi che la matrice di adiacenza per i grafi visti prima occupa una quantità di memoria molto superiore a quella realmente richiesta. Infatti, per memorizzare le informazioni relative a 7 nodi e 9 archi, stiamo creando una matrice di 49 elementi. In effetti la scelta della matrice di adiacenza

come metodo di rappresentazione di un grafo dipende molto da quali sono le caratteristiche del grafo. Un grafo con molti nodi e molti archi "sfrutta" appieno la memoria allocata mentre un grafo con molti nodi e pochi archi porterà ad uno spreco inevitabile di memoria.

ESEMPIO

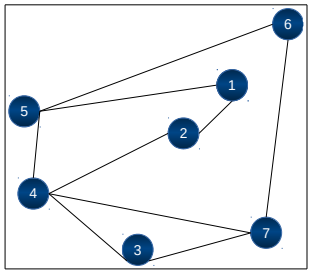


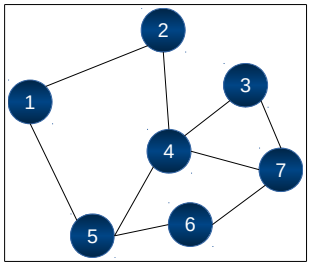
	1	**2**	**3**	**4**	**5**	**6**	**7**
1	0	1	0	1	1	0	1
2	1	0	1	1	1	0	0
3	0	0	1	1	0	0	1
4	1	1	1	0	1	0	1
5	1	1	0	1	0	0	1
6	0	0	0	1	1	0	1
7	1	0	1	1	0	1	0

Nel prossimo paragrafo si vedrà una rappresentazione alternativa alla matrice di adiacenza che risolve il problema dello spreco di memoria ma, come già anticipato, avrà come contraltare una complicazione delle operazioni di manipolazione del grafo.

3.2 Liste di adiacenza

Come alternativa alla matrice di adiacenza, è possibile rappresentare un grafo semplicemente elencando i nodi e, per ogni nodo, elencare quali sono quelli a cui è direttamente collegato.





1: 2, 5
2: 1, 4
3: 4, 7
4: 2, 3, 5, 7
5: 1, 4, 6
6: 5, 7
7: 3, 4, 6

È subito evidente che la quantità di informazione da memorizzare è molto più piccola rispetto al caso visto in precedenza. Qui, per memorizzare le stesse informazioni sui due grafi si sono utilizzati solo 18 elementi! Tuttavia, in maniera del tutto simmetrica rispetto al caso precedente, per verificare l'esistenza di un arco è necessario scorrere la lista dei nodi adiacenti. Per grafi con molti archi questo può significare un tempo non trascurabile e il vantaggio di occupare meno spazio in memoria viene compensato in negativo dalla maggiore quantità di tempo necessaria a reperire l'informazione. Ancora una volta, la scelta del tipo di rappresentazione non è assoluta ma dipende dalle caratteristiche particolari del grafo.

La creazione di un grafo mediante liste di adiacenza può essere implementata

in linguaggio C++ nel seguente modo[3]

```
list<int>* creaGrafoLista(int N)
{
        list<int>* L = new list<int>[N];

        while (true)
        {
                int i,j;
                printf("Inserisci una coppia di nodi separati da uno spazio: ");
                scanf("%i %i", &i, &j);

                //se l'indice è negativo interrompo il caricamento dei dati
                if (i<0 || j < 0)
                        break;
                L[i].push_front(j);
                L[j].push_front(i);

        }

        return L;

}
```

3.3 Esercizi

1. Dati i grafi dell'esercizio al capitolo precedente, indicare la matrice di adiacenza e le liste di adiacenza
2. Date le seguenti strutture dati, disegnare i grafi ad esse associati

	1	**2**	**3**	**4**	**5**	**6**	**7**
1	0	1	0	1	0	0	1
2	1	0	1	1	1	0	0
3	0	0	1	1	0	0	1
4	1	1	1	0	1	0	1
5	1	0	0	0	0	0	1
6	0	0	0	1	1	0	1
7	1	0	0	0	0	1	0

1:2,5,4
2:1,4,6
3:4,7,2
4:2,5,7
5:1,4,6
6:5,1,7
7:3,1,6

3. Scrivere una funzione che verifichi l'esistenza di un arco in un grafo sia nel caso di rappresentazione mediante liste di adiacenza che di matrice di adiacenza. (Es*empio: un sito di trasporti utilizza questa funzione per verificare che siano connessioni dirette fra due località.)*
4. Scrivere una funzione che inserisca un nuovo arco nel grafo, utilizzando sia matrice di adiacenza che liste di adiacenza. *(Esempio: l'inserimento di un arco in un grafo avviene, ad esempio, quando si crea una nuova connessione su un social network.)*
5. Modificare il codici per la creazione delle liste di adiacenza in modo che se un arco risulta già inserito, non venga duplicato. *(Esempio: quando un motore di ricerca crea il grafo dei link di una pagina web, i collegamenti multipli ad una stessa pagina non sono tenuti in considerazione.)*

3In questa implementazione si fa uso delle librerie standard C++ per evitare di appesantire il codice con la costruzione esplicita della lista, come ad esempio riportato in [Sedgewick]

4 Ricerca

Tempo previsto: 2 ore di lezione frontale + 4 ore di laboratorio

Come già accennato in precedenza, il punto critico nella comprensione dei grafi consiste nel passare da una visione che potremmo chiamare informale (o visiva) ad una visione algoritmica (o strutturata). Già nel capitolo 2 si sono introdotti alcuni elementi che dovrebbero guidare verso una visione più generale del concetto di grafo (definizione di cammino, definizione di grafo) e nel capitolo 3 si sono definiti i metodi di rappresentazione (matrice e liste di adiacenza). Dopo aver saltato l'ostacolo relativo alla rappresentazione, rimane da capire come utilizzare questa rappresentazione per eseguire delle operazioni che eseguite con carta e penna risultano estremamente semplici. Se ad esempio pensiamo ad un cammino su un grafo, risulta immediato associarlo a degli archi colorati, succede ad esempio quando impostiamo la rotta sul navigatore e ci vengono mostrate le strade da percorrere.

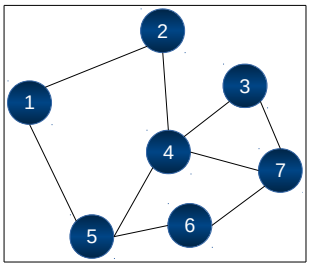
Per capire come utilizzare i grafi, riprendiamo alcune delle domande che ci eravamo posti nel capitolo 1 e proviamo a riscriverle utilizzando la terminologia appropriata

1. indagare se esiste una relazione indiretta fra due elementi del grafo
   - **esiste un cammino fra due nodi?**
2. sapere quali sono gli elementi del grafo che hanno una interdipendenza reciproca
   - **quali sono le componenti connesse del grafo?**

Avendo a disposizione l'immagine del grafo, la risposta a queste domande è tutto sommato semplice. Ma, come già detto in precedenza, per un calcolatore un grafo è qualcosa di diverso da un'immagine e quindi c'è bisogno di un algoritmo che possa essere applicato alle strutture dati con cui rappresentiamo il grafo. Tale algoritmo deve fornire uno strumento per l'analisi del grafo e può basarsi solo informazioni locali, cioè sulla conoscenza dei nodi adiacenti ad un nodo dato.

Nell'ambito dei grafi si parla di *algoritmo di visita* di un grafo per indicare un procedimento mediante il quale riusciamo ad analizzare tutte le componenti del grafo in maniera *sistematica*.

Consideriamo il solito grafo e le sue liste di adiacenza



1:2,5
2:1,4
3:4,7
4:2,3,5,7
5:1,4,6
6:5,7
7:3,4,6

Un generico algoritmo di visita di un grafo può essere espresso in termini generali mediante lo pseudocodice seguente

```
scegli un nodo di partenza e aggiungilo alla lista dei nodi da visitare
fino a che non sono stati visitati tutti i nodi
        scegli il nodo corrente
        visita il nodo corrente
        rimuovi il nodo corrente dalla lista e marcalo come visitato
        scegli i prossimi nodi da visitare
        aggiungi questi nodi alla lista dei nodi da visitare
```

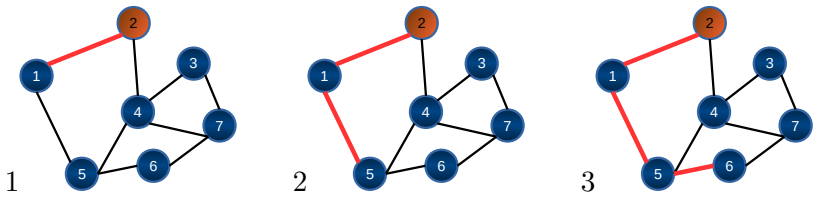
Partendo da un nodo qualsiasi, possiamo seguire i collegamenti e muoverci lungo di essi mantenendo traccia dei nodi su cui transitiamo. Ad esempio, partendo dal nodo 2 potremmo andare sul nodo 1 e poi sul nodo 4. Poi potremmo ripartire dal nodo 1 e visitare il 5. Dal 5 passare al 6, poi dal 4 passare al 3 e poi al 7. Avremmo visitato il grafo nel seguente ordine:

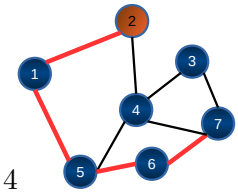
2, 1, 4, 5, 6, 3, 7

L'ordine con cui abbiamo seguito i collegamenti è stato del tutto arbitrario e quindi difficilmente questo procedimento può essere codificato in un algoritmo. Affinché possiamo scrivere un procedimento standard per la visita di un grafo diventa imprescindibile decidere in che *ordine* vengono visitati i nodi. Ed è proprio l'ordine con cui seguiamo i collegamenti del grafo, cioè *come scegliamo* il prossimo nodo da visitare, che differenzia gli algoritmi di visita e ne costituisce il punto principale.

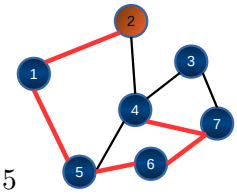
4.1 Visita in profondità

Sempre facendo riferimento al grafo precedente, proviamo a vedere cosa succede se decidiamo di visitare i nodi scegliendo tra quelli immediatamente adiacenti al nodo in cui ci troviamo. Una possibile evoluzione è la seguente:

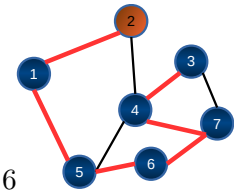




4



5



6

L'ordine di visita a partire dal nodo 2 sarà

1, 5, 6, 7, 3

Questa è *una* delle possibilità in quanto sia al passo 1 che al passo 4 eravamo di fronte a due possibili scelte. Ovviamente se avessimo scelto un nodo alternativo, l'evoluzione sarebbe stata diversa. Questo piccolo esperimento ci permette di fare un'osservazione molto importante: poiché tutti i nodi adiacenti possono essere scelti come prossimo nodo, è necessario restringere le possibilità di scelta in modo da rendere il procedimento ripetibile. Per fare ciò è possibile scegliere i nodi da visitare basandosi sull'ordine assegnato al momento della creazione delle liste di adiacenza. Nel caso del grafo in esame, partendo dal nodo 2 si visiteranno nell'ordine

1, 5, 4, 3, 7, 6

A questo punto, possiamo fare due considerazioni

1) l'ordine di visita dipende dall'ordine di inserimento dei nodi nella lista, anche se tutti i nodi hanno “pari dignità”
2) una volta fissata la rappresentazione del grafo, l'applicazione dell'algoritmo è deterministica, ossia l'ordine con cui vengono visitati i nodi è sempre lo stesso

Il criterio di visita appena introdotto dà luogo a quella che viene chiamata visita in profondità o *Depth First Search (DFS)* e può essere implementata in C++. Nello spezzone di codice che segue i commenti si riferiscono alle fasi di visita espresse in pseudocodice.

```
void DFScallIterativa(list<int>* G, int nodo, bool* visitato, int N)
{
	//crea la lista dei nodi da visitare
	list<int> prossimo;
	prossimo.push_front(nodo);

	//finchè non sono stati visitati tutti i nodi
	while(!prossimo.empty())
	{
		//scegli un nodo di partenza e marcalo come nodo corrente
		nodo = prossimo.front();
		//rimuovi il nodo corrente dalla lista...
		prossimo.pop_front();

		if(!visitato[nodo])
		{
			//… e marcalo come visitato
			visitato[nodo] = true;

			cout << " sto visitando: " << nodo << "\n";
```

```
                        //scegli i prossimi nodi da visitare
                        list<int>::reverse_iterator i;
                        for(i=G[nodo].rbegin(); i != G[nodo].rend(); ++i)
                                if(!visitato[*i])
                                        //aggiungi questi nodi
                                        //alla lista dei nodi da visitare
                                        prossimo.push_front(*i);
                }
        }

}
void DFSIterativa(list<int>* G, int nodoInizio, int N)
{
        bool* visitato = new bool[N];
        for (int i = 0; i < N; i++)
                visitato[i] = false;
        for (int i = 0; i <N; i++)
                if(!visitato[i])
                        DFScallIterativa(G, i, visitato, N);
        delete visitato;
}
```

Le istruzioni che caratterizzano la DFS sono le seguenti:

```
nodo = prossimo.front();
prossimo.pop_front();
```

```
for(i=G[nodo].rbegin(); i != G[nodo].rend(); ++i)
                                if(!visitato[*i])
                                        //aggiungi questi nodi
                                        //alla lista dei nodi da visitare
                                        prossimo.push_front(*i);
```

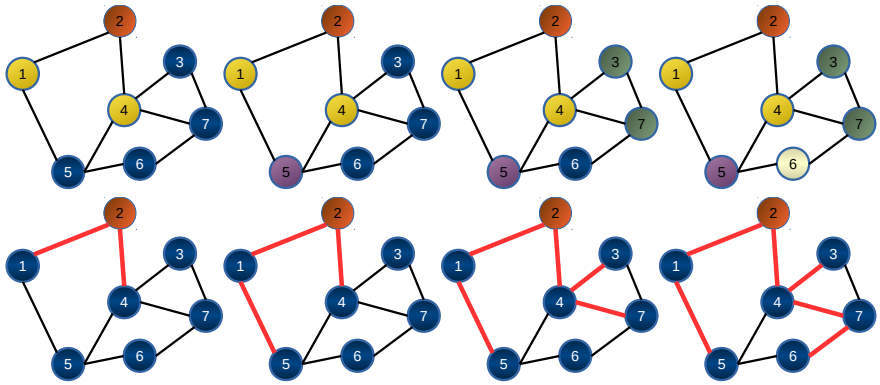
I nodi da visitare vengono scelti inserendo nella lista i nodi adiacenti al nodo corrente. All'iterazione successiva, viene prelevato dalla lista l'ultimo nodo inserito che corrisponde al primo nodo della lista degli adiacenti. La lista viene cioè gestita come una pila[4] e di conseguenza l'ultimo nodo inserito è il primo ad essere visitato all'iterazione seguente, realizzando così la scelta di visitare il nodo immediatamente adiacente al nodo corrente.

La particolarità della DFS è di partire da un nodo e proseguire fino alla massima distanza possibile prima di passare alla visita degli altri nodi adiacenti al nodo corrente.

4.2 Visita in ampiezza

Seguendo lo stesso ragionamento che abbiamo applicato per la DFS, proviamo a vedere cosa succede se scegliamo di visitare prima *tutti* i nodi adiacenti al nodo corrente. Sempre partendo dal nodo 2, vediamo quale può essere l'evoluzione della visita del grafo (Nelle figure seguenti sono marcati con lo stesso colore i nodi tra i quali scegliere il prossimi da visitare). Si visitano prima i nodi 1 e 4, tutti adiacenti al nodo 2. Successivamente si scelgono gli adiacenti di 1, quindi il nodo 5, e di 4, quindi i nodi 3 e 7. A partire da quest'ultimo rimane da visitare il 6.

4La conoscenza della struttura dati pila è un prerequisito dell'unità didattica.



Un possibile ordine di visita è il seguente

1, 4, 5, 3, 7, 6.

Come era già successo precedentemente, se al passo 2 avessimo effettuato una scelta diversa e avessimo deciso di visitare prima gli adiacenti di 4, avremmo potuto avere come sequenza di visita la seguente

1, 4, 7, 3, 5,6

Anche in questo caso necessitiamo di decidere una regola standard per l'inserimento in lista dei nodi da visitare. Ed anche in questo caso la soluzione più semplice è quella di inserire i nodi nell'ordine con cui sono stati memorizzati al momento della creazione della struttura.

Adottando come criterio di scelta dei nodi tutti gli adiacenti del nodo corrente si realizza la cosiddetta visita in ampiezza o *Breadth First Searh (BFS)* la cui implementazione in C++ è di seguito riportata.

```
void BFScallIterativa(list<int>* G, int nodo, bool* visitato, int N)
{
	//crea la lista dei nodi da visitare
	list<int> prossimo;
	prossimo.push_front(nodo);

	//finchè non sono stati visitati tutti i nodi
	while(!prossimo.empty())
	{
		//scegli un nodo di partenza e marcalo come nodo corrente
		nodo = prossimo.back();
		//rimuovi il nodo corrente dalla lista...
		prossimo.pop_back();

		if(!visitato[nodo])
		{
			//… e marcalo come visitato
			visitato[nodo] = true;

			cout << " sto visitando: " << nodo << "\n";

			//scegli i prossimi nodi da visitare
			list<int>::reverse_iterator i;
			for(i=G[nodo].rbegin(); i != G[nodo].rend(); ++i)
				if(!visitato[*i])
					//aggiungi questi nodi
					//alla lista dei nodi da visitare
```

```
                                        prossimo.push_front(*i);
                }
        }

}
void BFSIterativa(list<int>* G, int nodoInizio, int N)
{
        bool* visitato = new bool[N];
        for (int i = 0; i < N; i++)
                visitato[i] = false;
        for (int i = 0; i <N; i++)
                if(!visitato[i])
                        BFScallIterativa(G, i, visitato, N);
        delete visitato;
}
```

I nodi da visitare vengono scelti inserendo nella lista i nodi adiacenti al nodo corrente. All'iterazione successiva, viene prelevato dalla lista il primo nodo inserito che corrisponde all'ultimo nodo della lista degli adiacenti. La lista viene cioè gestita come una coda[5] e di conseguenza il primo nodo inserito è il primo ad essere visitato all'iterazione seguente, realizzando così la scelta di visitare prima i nodi adiacenti al nodo corrente.

Le istruzioni che caratterizzano la BFS sono le seguenti:

```
nodo = prossimo.front();
prossimo.pop_back();
```

```
for(i=G[nodo].rbegin(); i != G[nodo].rend(); ++i)
                                if(!visitato[*i])
                                        //aggiungi questi nodi
                                        //alla lista dei nodi da visitare
                                        prossimo.push_front(*i);
```

La particolarità della BFS è di partire da un nodo e visita i nodi “per livelli”.

4.3 Considerazioni finali

DFS e BFS realizzano un attraversamento del grafo facendo uso di informazione locale. In entrambi i casi, il cammino viene “costruito” basandosi sui nodi adiacenti del nodo corrente e in entrambi i casi il cammino dipende dall'ordine in cui sono memorizzati i dati nella lista di adiacenza. Ciò che differenzia i due metodi di visita è la diversa gestione della struttura dati di appoggio che viene utilizzata: una pila (LIFO)nel caso di DFS, una coda (FIFO) nel caso di BFS.

L'aver definito un algoritmo di visita permette di rispondere ad entrambe le domande formulate all'inizio del capitolo. Infatti, per verificare l'esistenza di un cammino fra un nodo A ed un nodo B è sufficiente far iniziare la ricerca dal nodo A e arrestare l'esecuzione dell'algoritmo nel momento in cui il nodo B viene visitato.

Esercizio

A partire dalle implementazioni di DFS e BFS, scrivere una funzione che

5La conoscenza della struttura dati pila è un prerequisito dell'unità didattica.

verifichi l'esistenza di un cammino da un nodo sorgente ad un nodo destinazione. (suggerimento: qual è la prima condizione da verificare per rispondere immediatamente alla domanda nel caso più semplice?)

In maniera del tutto simile è possibile rispondere alla seconda domanda. Le componenti connesse del grafo possono essere individuate memorizzando di volta in volta tutti i nodi visitati.

Esercizio

A partire dalle implementazioni di DFS e BFS, scrivere una funzione che restituisca le componenti connesse di un grafo

Infine è possibile estendere i due algoritmi in modo che oltre alle informazioni sui nodi, vengano restituite informazioni sugli archi che collegano i nodi durante la visita. Per fare ciò è sufficiente memorizzare per ogni nodo da aggiungere alla lista dei non visitati l'informazione sul nodo corrente (detto *predecessore*).

Esercizio

Modificare BFS e DFS in modo che venga restituita la lista dei predecessori dei nodi visitati. Scrivere poi una funzione che data la lista dei predecessori costruisca il grafo dei nodi visitati. Qual è la caratteristica di questo grafo?

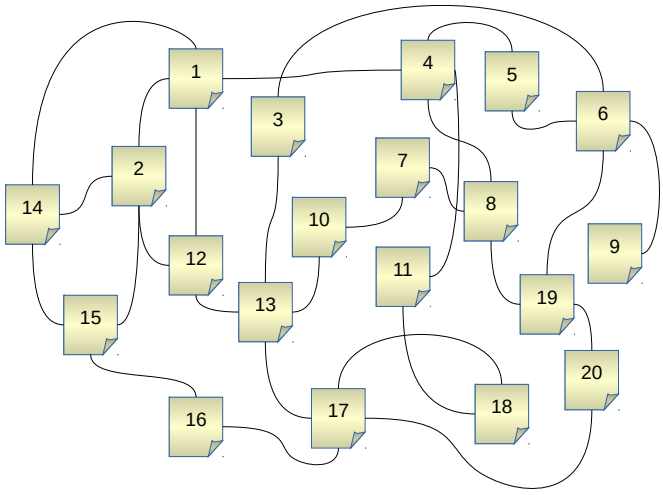
4.4 Esempi conclusivi

Tempo previsto: 1 ora di lezione frontale

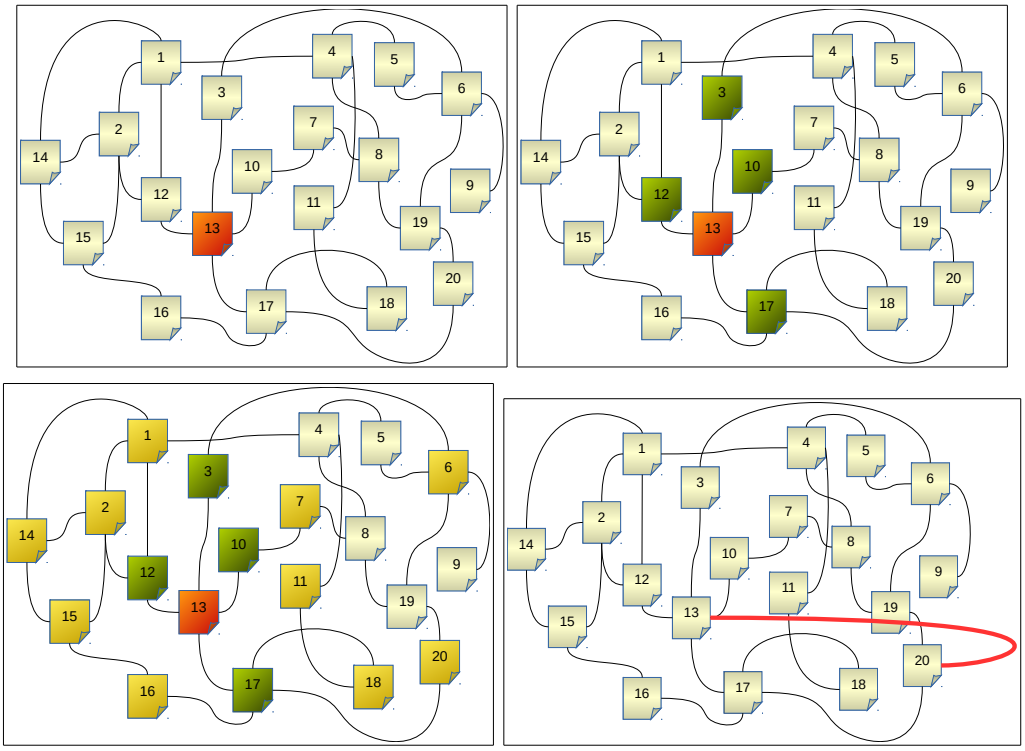
Negli ultimi due paragrafi vengono mostrati due esempi concreti di applicazioni che fanno uso dei grafi. Nel primo esempio viene spiegato, in maniera molto semplificata, il meccanismo che è alla base dei suggerimenti delle amicizie su facebook. In questo caso si fa uso dei cammini per introdurre il concetto di *chiusura transitiva*. Nel secondo esempio viene mostrato in che modo i programmi di fotoritocco riescono a selezionare un'area omogenea di un'immagine mediante lo strumento generalmente chiamato "bacchetta magica". Questa volta lo strumento utilizzato è la componente connessa.

4.4.1 Chiusura transitiva (o Come Facebook suggerisce le amicizie)

Consideriamo il seguente grafo che rappresenta le relazioni di conoscenza fra gli utenti di un social network



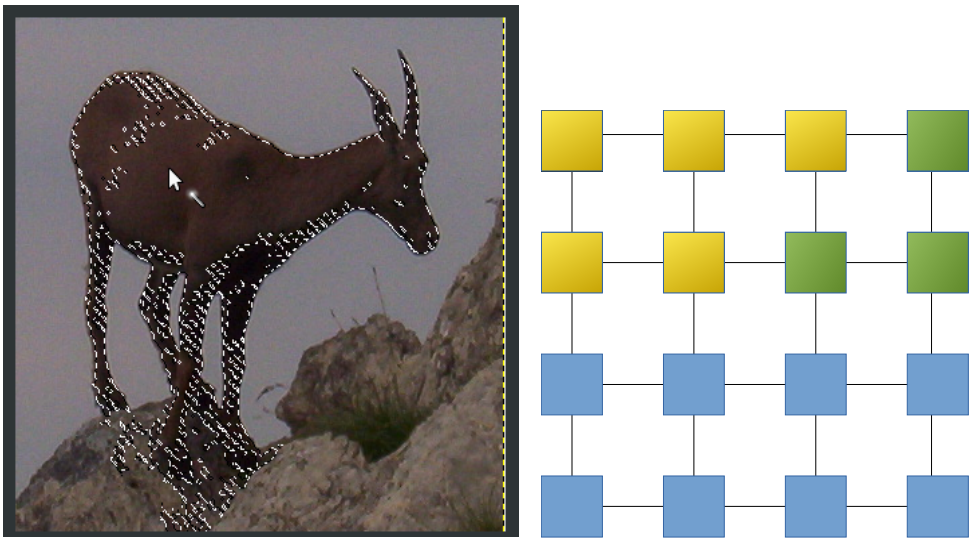
Alla luce di quanto visto fino ad ora, vediamo qual è la logica che sta alla base delle funzionalità di suggerimento delle connessioni dei social network. Proviamo ad applicare una BFS a partire dal nodo 13 e ci fermiamo quando raggiungiamo il secondo livello.



Quelli evidenziati in giallo sono i potenziali contatti a cui, secondo il software di gestione del social, potremmo essere interessati in quanto a noi più "vicini". Infatti, i nodi marcati con lo stesso colore hanno la proprietà di essere alla stessa distanza (intesa come numero di archi da attraversare) dal nodo iniziale. Supponiamo ora che il profilo del nodo 20 sia per noi realmente interessante: visto che esiste un cammino fra il nodo 13 ed il nodo 20 inseriamo nel grafo un arco che li collega direttamente. Ripetendo questo procedimento per tutti i nodi marcati in giallo otteniamo la chiusura transitiva del nodo 13 con tutti i nodi a distanza 2.

4.4.2 Flood fill (o Come selezioniamo un'area omogenea sui programmi di fotoritocco)

Uno degli strumenti più utilizzati nell'ambito del fotoritocco è la cosiddetta bacchetta magica. È uno strumento che permette di selezionare aree omogenee di un'immagine, tipicamente quelle con lo stesso colore (o comunque colori molto simili tra di loro. L'immagine seguente mostra un'area delimitata da una linea tratteggiata la cui tonalità è decisamente sul marrone.



Cerchiamo di capire come funziona la bacchetta magica basandoci sulle nostre conoscenze sui grafi.

Un'immagine è formata da pixel, ad ognuno dei quali è associato un colore. I pixel possono essere considerati come nodi di un grafo. Ogni pixel è "collegato" al suo vicino. Quando con la bacchetta magica selezioniamo un pixel viene effettuata una visita del grafo (BFS o DFS) che prosegue fino a quando il nodo adiacente a quello corrente ha lo stesso colore (o comunque un colore "abbastanza simile").