3. Ottimizzazione Combinatoria con Mathematica

Marco Liverani

Università degli Studi Roma Tre
Dipartimento di Matematica e Fisica
Corso di Laurea in Matematica
E-mail liverani@mat.uniroma3.it

Marzo 2014

Revisione del 15 marzo 2014.

1 Premessa

La verifica delle proprietà e del comportamento di determinate classi di grafi sottoposte ad operazioni specifiche spesso può trovare un utile supporto anche nella realizzazione di semplici programmi in grado di eseguire rapidamente controlli su istanze del problema oggetto dello studio che manualmente risulterebbero ingestibili. Per questo scopo il software Mathematica, con il package denominato `DiscreteMath::Combinatorica`, costituisce un ambiente estremamente potente e confortevole, alternativo alla realizzazione di programmi in linguaggio C o con altri linguaggi di programmazione di alto livello. Il pacchetto estende il sistema di base con oltre 450 funzioni di combinatoria e teoria dei grafi. Include funzioni per costruire grafi ed altri oggetti combinatori, per calcolare numerosi invarianti su tali oggetti e naturalmente anche per visualizzarli in una forma (grafica o numerica) estremamente efficace.

Gli studenti dell'Università Roma Tre, grazie ad una convenzione stipulata tra l'Ateneo e la Wolfram Research, azienda che produce il software Mathematica, possono acquisire gratuitamente una copia del prodotto pienamente funzionante. Le istruzioni per il download del software di installazione disponibile per numerose piattaforme (Microsoft Windows, Linux, Apple MacOS X, ecc.) sono disponibili sul sito Internet dell'Università all'indirizzo

`http://asi.uniroma3.it/page.php?page=convenzio4`

2 Richiami sull'uso di Mathematica

Mathematica mette a disposizione un ambiente interattivo con cui definire oggetti di tipo matematico ed eseguire calcoli utilizzando funzioni offerte dall'ambiente di base o da uno dei suoi *package* aggiuntivi, ovvero consentendo all'utente di costruire delle funzioni utilizzando un linguaggio di programmazione interno.

Per inserire un comando è sufficiente digitarlo e battere i tasti Shift+Enter; battendo soltanto il tasto Enter, senza tenere premuto contemporaneamente Shift, si manderà a capo il cursore, senza però comunicare a Mathematica che l'inserimento dell'istruzione da elaborare è terminato.

Ogni riga di input digitata dall'utente viene numerata progressivamente ed è identificata dal *prompt* «`In[`*n*`]`». Analogamente l'output prodotto dal sistema a fronte dell'input numero *n* viene identificato dalla *label* «`Out[`*n*`]`». Se non si desidera che l'esito del comando digitato in input venga visualizzato in output, allora basterà aggiungere il punto-e-virgola "`;`" al termine dell'istruzione inserita in input. È possibile utilizzare il simbolo di percentuale "`%`" per indicare il risultato dell'ultimo calcolo eseguito dal sistema. Ad esempio:

```
 In[1]:= 5+4
Out[1] = 9
 In[2]:= 4+3;
 In[3]:= %
Out[3] = 7
```

Tutti i calcoli vengono effettuati utilizzando la massima precisione possibile; per far questo, ad esempio, le frazioni vengono lasciate indicate in forma semplificata, ma senza ricorrere ad espressioni con numeri decimali. Se si desidera invece operare sui termini dell'espressione considerandoli come numeri decimali (non interi), allora basterà aggiungere il punto decimale subito dopo uno degli operandi; questo potrebbe portare anche a risultati approssimati. Ad esempio:

```
 In[1]:= 6/4
Out[1] = 3/2
```

```
 In[2]:= 6./4
Out[2] = 1.5
 In[3]:= 1/3
Out[3] = 1/3
 In[4]:= 1/3.
Out[4] = 0.3333
```

Per ottenere una approssimazione numerica del risultato di un'espressione si deve utilizzare la funzione `N` che consente anche di specificare il numero di cifre decimali a cui si deve spingere la rappresentazione del numero. Ad esempio:

```
 In[1]:= 1/3
Out[1] = 1/3
 In[2]:= N[1/3]
Out[2] = 0.333333
 In[3]:= N[1/3, 10]
Out[3] = 0.3333333333
```

Gli operatori aritmetici fondamentali sono gli stessi presenti in ogni linguaggio di programmazione, ma ve ne sono altri (il fattoriale, ad esempio, indicato con il simbolo "`!`") che generalmente invece non sono presenti in altri linguaggi. L'elevamento a potenza è indicato con il simbolo "`^`".

Nel definire un'espessione matematica si possono usare le parentesi tonde per esplicitare le priorità tra le diverse operazioni, così come si è abituati a fare normalmente. L'operazione di moltiplicazione può anche essere indicata in modo implicito, senza specificare il simbolo "`*`". Ad esempio si può scrivere «`(3+2)(5+4)`» invece del più classico (nei linguaggi per calcolatori) «`(3+2)*(5+4)`».

Tutte le funzioni definite nel sistema o nei package aggiuntivi sono rappresentate da nomi che iniziano con una lettera maiuscola, come ad esempio `Sin` per la funzione seno, `Sqrt` per la radice quadrata, ecc. Per avere delle indicazioni sull'uso di una determinata funzione si può specificare il nome della funzione preceduta dal simbolo del punto interrogativo; il sistema visualizzerà un breve messaggio di aiuto, seguito da un *link* con cui si potrà visualizzare la pagina del sistema di help on-line in cui viene descritta la funzione stessa:

```
 In[1]:= ?Sqrt
Out[1] = Sqrt[z] gives the square root of z. More...
```

Gli argomenti delle funzioni vengono indicati tra parentesi quadre, separati fra loro da virgole: per specificare gli argomenti di una funzione, quindi, non possono essere usate le parentesi tonde. Ad esempio:

```
 In[1]:= Cos[π]
Out[1] = -1
```

L'operatore di *assegnazione* "=" consente di attribuire il valore ad una variabile e, al tempo stesso, crearne un'istanza se questa non fosse già stata precedentemente creata. Esistono innumerevoli tipi di dati e strutture dati che vengono rese trasparenti all'utente: di fatto il tipo della variabile o dell'oggetto strutturato che si crea dipende dal contesto e dunque dal tipo di dato che viene assegnato. Come vedremo più avanti in questo modo è possibile creare semplici variabili scalari, liste, matrici, insiemi, grafi ed altro ancora. L'assegnazione del valore ad una variabile (o più in generale ad un oggetto strutturato più complesso) è "statica" se viene effettuata con l'operatore "=", mentre viene mantenuto il riferimento alla definizione (e dunque il valore varia dinamicamente) utilizzando l'operatore di *definizione* "`:=`"; un esempio chiarirà il meccanismo meglio di molte parole:

```
 In[1]:= x = 10
Out[1] = 10
```

```
 In[2]:= y = x
Out[2] = 10
 In[3]:= z:=x
 In[4]:= x = 2x
Out[4] = 20
 In[5]:= y
Out[5] = 10
 In[6]:= z
Out[6] = 20
```

Per azzerare una variabile o distruggere un oggetto complesso si può utilizzare la funzione `Clear`; se prima non si è utilizzata questa funzione non è possibile ridefinire il tipo di una variabile o di un oggetto.

Con l'operatore ":=" è possibile definire anche delle funzioni vere e proprie. Per indicare i simboli che rappresentano le variabili formali passate come argomento della funzione che si sta definendo, si deve indicare il nome della variabile, seguito dal simbolo *underscore* "_" ed eventualmente dal tipo di dato rappresentato dalla variabile stessa. La possibilità di definire funzioni è alla base della modalità con cui è possibile sviluppare procedure o programmi complessi con Mathematica. Vediamo un esempio elementare:

```
 In[1]:= f[x_, k_] := x^2 + k x
 In[2]:= f[3,5]
Out[2] = 24
```

Le parentesi graffe devono essere utilizzate per definire liste di elementi. Mathematica non distingue tra liste e vettori o array: gli elementi sono ordinati e identificati da un numero progressivo. Le matrici sono definite come liste di liste. La funzione `Part` serve a restituire un elemento o una parte di un vettore o di una matrice, mentre la funzione `Length` restituisce la lunghezza (il numero di elementi) di una lista. Ad esempio:

```
 In[1]:= x = {2, 3, 5, 7, 11}
Out[1] = {2, 3, 5, 7, 11}
 In[2]:= m = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
Out[2] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
 In[3]:= Part[m,2,3]
Out[3] = 6
 In[4]:= Length[x]
Out[4] = 5
```

L'istruzione numero 3 richiede di selezionare (con la funzione `Part`) l'elemento $m_{2,3}$ della seconda riga e terza colonna della matrice m.

Il package `DiscreteMath::Combinatorica` aggiunge numerose funzioni per la manipolazione di insiemi (rappresentati come liste e array) e grafi. Per caricare il pacchetto si deve impartire il seguente comando per Mathematica 5.x:[1]

```
 In[1]:= << DiscreteMath`Combinatorica`
```

Se si utilizza il software Mathematica 6.x il comando da impostare è invece il seguente:

```
 In[1]:= << Combinatorica`
```

[1] Il carattere di "apice" utilizzato nei comandi per il caricamento del pacchetto `Combinatorica` è il *backtick*, ossia l'accento presente sulle tastiere con il layout americano, in alto a sinistra; il carattere corrisponde al codice ASCII 96 e, su un PC in ambiente Windows, può essere ottenuto digitando il numero 96 sul tastierino numerico, tenendo premuto al tenpo stesso il tasto ALT.

A partire dalla versione 8 del software Mathematica, numerosi oggetti e funzioni di matematica discreta o di calcolo combinatorio sono stati inseriti nel "nucleo" del prodotto e dunque, per molte di queste operazioni, non è più richiesto il caricamento di pacchetti aggiuntivi come "Combinatorica". Tuttavia tale pacchetto contiene la definizione di alcune funzioni non implementate nel nucleo base del prodotto e quindi, in certi casi, è ancora necessario caricarlo.

3 Operazioni elementari sugli insiemi

Un insieme può essere definito specificandone gli elementi tra parentesi graffe (es.: `{1,2,3,4}`). Mathematica implementa le principali operazioni elementari sugli insiemi: la funzione `Union` produce l'unione fra due o più insiemi, la funzione `Intersection` genera l'intersezione e la funzione `Complement` applicata a due o più insiemi, fornisce gli elementi del primo insieme che non sono contenuti anche negli altri. Ad esempio:

```
 In[1]:= Union[{2, 4, 6}, {1, 3, 5, 7}]
Out[1] = {1, 2, 3, 4, 5, 6, 7}
 In[2]:= Intersection[{2, 4, 6}, {1, 2, 3, 4}]
Out[2] = {2, 4}
 In[3]:= Complement[{2, 4, 6}, {1, 2, 3}]
Out[3] = {4, 6}
```

L'operazione combinatoria più elementare offerta da `DiscreteMath::Combinatorica` è la costruzione delle permutazioni degli elementi di un insieme; questa operazione può essere effettuata con la funzione `Permutations`:

```
 In[1]:= Permutations[{1,2,3}]
Out[1] = {{1,2,3}, {1,3,2}, {2,1,3}, {2,3,1}, {3,1,2}, {3,2,1}}
```

Visto che nell'ambiente Mathematica gli insiemi vengono trattati come array, gli elementi hanno un ordine ben preciso nel momento in cui l'insieme stesso viene definito e dunque possono essere identificati da un indice: $A = \{a_1, a_2, \ldots, a_n\}$. Quindi una permutazione di un insieme A coincide con una permutazione dell'insieme degli indici dei suoi elementi $\{1, 2, \ldots, n\}$. È possibile ottenere una specifica permutazione degli elementi di un insieme, ad esempio $A = \{\text{cane}, \text{gatto}, \text{pesce}\}$, utilizzando la funzione `Permute` e specificando come argomento, oltre all'insieme di cui si vuole ottenere la permutazione, anche l'ordine con cui gli elementi devono comparire nella permutazione, utilizzando gli indici numerici degli elementi stessi. Ad esempio:

```
 In[1]:= Permute[{"cane", "gatto", "pesce"}, {2, 3, 1}]
Out[1] = {gatto, pesce, cane}
```

Naturalmente non tutte le sequenze di numeri interi rappresentano una permutazione: ad esempio l'insieme $\{1, 2, 1\}$ non rappresenta una permutazione perché il primo elemento dell'insieme (quello identificato dall'indice 1) sarebbe ripetuto due volte, ma anche l'insieme $\{1, 2, 4\}$ non rappresenta una permutazione, perché o l'insieme è costituito da soli 3 elementi e allora non esiste un elemento di indice 4, oppure è composto da 4 elementi e allora nella permutazione manca l'elemento di indice 3. Con la funzione booleana `PermutationQ`[2] è possibile stabilire se un insieme di interi rappresenta o meno una permutazione: la funzione restituisce il valore logico *vero* (`True`) se l'insieme di n elementi è costituito da una permutazione dei naturali $1, \ldots, n$, mentre restituisce *falso* (`False`) altrimenti. Ad esempio:

[2] In generale le funzioni identificate da un nome che termina con la lettera "Q" sono funzioni booleane, che restituiscono quindi valore *vero* o *falso*.

```
 In[1]:= PermutationQ[{1,3,2}]
Out[1] = True
 In[2]:= PermutationQ[{2,5,7}]
Out[2] = False
```

La funzione `RandomPermutation` accetta come argomento un numero naturale n e produce una permutazione casuale dei primi n numeri naturali:

```
 In[1]:= RandomPermutation[3]
Out[1] = {2, 3, 1}
```

Naturalmente le funzioni possono essere anche composte fra di loro, come nel seguente esempio:

```
 In[1]:= Permute[{"cane", "gatto", "topo"}, RandomPermutation[3]]
Out[1] = {topo, gatto, cane}
```

Sono disponibili anche diverse funzioni per l'ordinamento degli elementi di un insieme: `Sort`, `SelectionSort`, `HeapSort`. Ad esempio:

```
 In[1]:= Sort[{4,2,3,1}]
Out[1] = {1, 2, 3, 4}
```

L'insieme delle parti di un insieme A, ossia l'insieme di tutti i sottoinsiemi di A, può essere facilmente ottenuto con la funzione `Subsets`. Inoltre, nella terminologia di Mathematica, un k-sottoinsieme di A è un sottoinsieme con esattamente k elementi; la famiglia di tutti i k sottoinsiemi di un insieme A, per un valore di k fissato, può essere ottenuta con la funzione `KSubsets`:

```
 In[1]:= Subsets[{1, 2, 3}]
Out[1] = {{}, {3}, {2, 3}, {2}, {1, 2}, {1, 2, 3}, {1, 3}, {1}}
 In[2]:= KSubsets[{1,2,3,4}, 3]
Out[2] = {{1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4}}
```

L'istruzione numero 2 richiede di generare tutti i sottoinsiemi di 3 elementi dell'insieme $\{1,2,3,4\}$. La stessa funzione `Subsets` richiamata passandogli come argomento un numero intero positivo n, genera la famiglia dei sottoinsiemi di $\{1,2,\ldots,n\}$:

```
 In[1]:= Subsets[3]
Out[1] = {{}, {3}, {2, 3}, {2}, {1, 2}, {1, 2, 3}, {1, 3}, {1}}
```

La funzione `RandomSubset` applicata all'insieme A restituisce un elemento scelto a caso nell'insieme delle parti di A:

```
 In[1]:= RandomSubset[{1,2,3,4,5}]
Out[1] = {2, 3, 5}
 In[2]:= RandomSubset[{1,2,3,4,5}]
Out[2] = {3, 4}
```

Con la funzione `Partitions` applicata all'intero $n > 0$ si ottengono tutte le possibili scomposizioni di n in $k \leq n$ naturali la cui somma è n:

$$\texttt{Partitions}(n) = \{\{a_{i_1},\ldots,a_{i_k}\} : \sum_{j=1}^{k} a_{i_j} = n\}$$

Ad esempio:

```
 In[1]:= Partitions[4]
Out[1] = {{4}, {3, 1}, {2, 2}, {2, 1, 1}, {1, 1, 1, 1}}
```

La funzione `Compositions` restituisce tutti gli insiemi di k interi positivi la cui somma sia n: `Compositions`$(n,k) = \left\{\{a_{i_1},\ldots,a_{i_k}\} : \sum_{j=1}^{k} a_{i_j} = n\right\}$. Dunque non vengono prodotte tutte le "partizioni" di n, come con la funzione `Partitions`, ma soltanto quelle costituite da insiemi di cardinalità k. Ad esempio:

```
 In[1]:= Compositions[4,2]
Out[1] = {{0, 4}, {1, 3}, {2, 2}, {3, 1}, {4, 0}}
```

Una partizione di un insieme A è una famiglia $\{A_1,A_2,\ldots,A_k\}$ di sottoinsiemi disgiunti di A, tale che la loro unione coincida con A; in altri termini possiamo scrivere: $A_1,A_2,\ldots,A_k \subseteq A$ tali che $A_i \cap A_j = \emptyset$ per ogni $i \neq j$ e $A_1 \cup \cdots \cup A_k = A$. La funzione `SetPartitions` applicata ad un insieme A restituisce la famiglia di tutte le partizioni di A; la stessa funzione può essere richiamata passandogli come argomento un numero naturale n, anziché un insieme: in questo caso restituisce tutte le partizioni dell'insieme $\{1,2,\ldots,n\}$. La funzione `KSetPartitions`, a cui si deve passare come argomento, oltre all'insieme da partizionare, anche un numero naturale k, restituisce tutte le partizioni in k sottoinsiemi. Vediamo alcuni esempi:

```
 In[1]:= SetPartitions[{a,b,c}]
Out[1] = {{{a, b, c}}, {{a}, {b, c}}, {{a, b}, {c}},
          {{a, c}, {b}}, {{a}, {b}, {c}}}
 In[2]:= SetPartitions[3]
Out[2] = {{{1, 2, 3}}, {{1}, {2, 3}}, {{1, 2}, {3}},
          {{1, 3}, {2}}, {{1}, {2}, {3}}}
 In[3]:= KSetPartitions[{a,b,c}, 2]
Out[3] = {{{a}, {b, c}}, {{a, b}, {c}}, {{a, c}, {b}}}
```

4 Costruzione e rappresentazione di grafi

Sono numerosissime le funzioni disponibili nel package `DiscreteMath::Combinatoria` che consentono di operare su grafi ed alberi. Possiamo suddividerle in alcuni gruppi omogenei: funzioni per la creazione di grafi, per l'estrazione di informazioni sui grafi, per l'esecuzione di operazioni di composizione sui grafi ed infine per la rappresentazione grafica di grafi ed alberi.

Per creare un nuovo grafo, o meglio, per definire un oggetto di tipo grafo, è possibile iniziare utilizzando la funzione `EmptyGraph` che consente di creare un grafo non orientato di $n \geq 0$ vertici privo di spigoli. Con le funzioni `AddVertex` e `AddEdge` è possibile aggiungere un vertice o uno spigolo al grafo. I vertici sono numerati progressivamente in modo automatico man mano che vengono aggiunti al grafo, a partire dal vertice etichettato con il numero 1. È possibile anche aggiungere n vertici contemporaneamente al grafo, con la funzione `AddVertices`, oppure aggiungere più di uno spigolo per volta con la funzione `AddEdges`.

```
 In[1]:= G = EmptyGraph[0]
Out[1] = -Graph:<0, 0, Undirected>-
 In[2]:= G = AddVertex[G]
Out[2] = -Graph:<0, 1, Undirected>-
 In[3]:= G = AddVertex[G]
Out[3] = -Graph:<0, 2, Undirected>-
 In[4]:= G = AddEdge[G, {1,2}]
Out[4] = -Graph:<1, 2, Undirected>-
 In[5]:= G = AddVertices[G,3]
Out[5] = -Graph:<1, 5, Undirected>-
 In[6]:= G = AddEdges[G, {{1,3}, {2,4}, {2,5}}]
Out[6] = -Graph:<4, 5, Undirected>-
```

L'istruzione numero 5 aggiunge tre nuovi vertici al grafo, mentre l'istruzione numero 6 aggiunge tre spigoli. In questi casi come risposta il sistema presenta un riepilogo di ciò che è stato creato con l'istruzione fornita in input: "-Graph:<*m*, *n*, Undirected>" indica che è stato costruito un grafo non orientato con *m* spigoli ed *n* vertici. Il numero di vertici e di spigoli di un grafo può essere ottenuto con le funzioni "V" e "M", rispettivamente; proseguendo l'esempio precedente si otterrebbe:

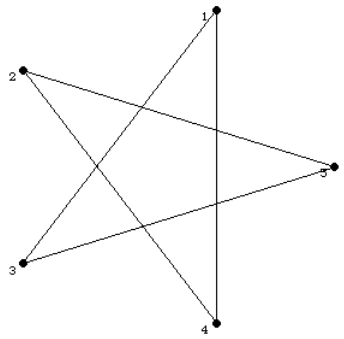
```
 In[7]:= V[G]
Out[7] = 5
 In[8]:= M[G]
Out[8] = 4
```

è possibile anche rimuovere vertici e spigoli con le funzioni DeleteVertices e DeleteEdges:

```
 In[9]:= G = DeleteVertices[G, {1,3}]
Out[9] = -Graph<2, 3, Undirected>-
```

La lista degli spigoli del grafo *G* può essere ottenuto con la funzione Edges, mentre la funzione Vertices visualizza la lista delle coordinate cartesiane dei vertici del grafo, nella sua rappresentazione grafica nel piano. è possibile infatti visualizzare il grafo utilizzando la funzione ShowGraph; tale funzione ammette l'opzione VertexNumber->On che consente di visualizzare l'etichetta numerica accanto ad ogni vertice del grafo. Ad esempio:

```
 In[1]:= G = EmptyGraph[5];
 In[2]:= G = AddEdges[G, {{1,3}, {1,4}, {2,4}, {3,5}, {2,5}}];
 In[3]:= ShowGraph[G, VertexNumber->On]
```




```
Out[3] = -Graphics-
```

Per impostare una volta per tutte le opzioni di visualizzazione di un determinato grafo, senza così doverle specificare ogni volta nel richiamare la funzione ShowGraph, si deve utilizzare la funzione SetGraphOptions:

```
 In[1]:= G=SetGraphOptions[G, VertexNumber->On];
```

Altre opzioni per la visualizzazione dei grafi sono riportate in Tabella 1 insieme ai possibili valori attribuibili ad ognuna delle opzioni.

Una volta costruito un grafo è possibile ottenerne alcune rappresentazioni non grafiche tipiche delle strutture dati utilizzate in informatica: liste di adiacenza (una lista per ogni vertice del grafo in cui siano riportati i vertici ad esso adiacenti) e matrici di adiacenza (matrici quadrate di ordine $|V(G)| \times |V(G)|$, il cui elemento $M_{i,j}$ è 1 se $(i, j) \in E(G)$ e zero altrimenti). Per far questo è possibile utilizzare le funzioni ToAdjacencyList e ToAdjacencyMatrix rispettivamente. In particolare quest'ultima funzione può essere utilizzata composta con la funzione MatrixForm per ottenere una rappresentazione matriciale classica:

Proprietà	Opzione	Valori
Orientazione spigoli	`EdgeDirection`	On, Off
Colore degli spigoli	`EdgeColor`	Blue, Red, Green, Black, Purple, Yellow
Colore dei vertici	`VertexColor`	Blue, Red, Green, Black, Purple, Yellow
Numerazione vertici	`VertexNumber`	On, Off
Colore dei numeri	`VertexNumberColor`	Blue, Red, Green, Black, Purple, Yellow
Stile dei vertici	`VertexStyle`	Disk, Disk[Large], Disk[n]
Colore dello sfondo	`Background`	Blue, Red, Green, Black, Purple, Yellow

Tabella 1: Alcune opzioni per la rappresentazione dei grafi

```
 In[1]:= G = EmptyGraph[4];
 In[2]:= G = AddEdges[G, {{1,3}, {1,4}, {2,4}}];
 In[3]:= ToAdjacencyLists[G]
Out[3] = {{3, 4}, {4}, {1}, {1, 2}}
 In[4]:= MatrixForm[ToAdjacencyMatrix[G]]
Out[4]//MatrixForm =
```

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

è possibile costruire un grafo a partire da una matrice di adiacenza o da una lista di adiacenza utilizzando le funzioni `FromAdjacencyMatrix` e `FromAdjacencyLists`; l'opzione `Type`, che può assumere i valori `Directed` o `Undirected`, consente di stabilire se gli spigoli devono essere orientati oppure no.

```
 In[1]:= G=FromAdjacencyLists[{{2},{3},{},{1,2}}, Type->Directed]
Out[1] =  -Graph:<4, 4, Directed>-
```

Utilizzando queste funzioni è possibile anche salvare su file la matrice di adiacenza o le liste di adiacenza di un grafo, oppure eseguire l'operazione inversa e caricare da file i valori della matrice di adiacenza o delle liste di adiacenza di un grafo. Con la seguente istruzione vengono salvate sui file "`liste_grafo.dat`" e "`matrice_grafo.dat`" rispettivamente le liste di adiacenza e la matrice di adiacenza di un grafo G definito precedentemente (la funzione `FilePrint` stampa il contenuto di un file):

```
 In[1]:= Export["liste_grafo.dat", ToAdjacencyLists[G]]
Out[1] =  liste_grafo.dat
 In[2]:= FilePrint["liste_grafo.dat"]
3   4   5
3   4
1   2   4   5
1   2   3
1   3
 In[3]:= Export["matrice_grafo.dat", ToAdjacencyLists[G]]
Out[3] =  liste_grafo.dat
 In[4]:= FilePrint["matrice_grafo.dat"]
0   0   1   1   1
0   0   1   1   0
1   1   0   1   1
1   1   1   0   0
```

```
1   0   1   0   0
 In[5]:= H = FromAdjacencyMatrix[Import["matrice_grafo.dat"]]
Out[5]= - Graph:<7,5,Undirected>-
```

Per costruire grafi più rapidamente è possibile partire da un grafo "noto", per la cui costruzione il package `DiscreteMath::Combinatorica` mette a disposizione delle funzioni dirette. Di seguito ne elenchiamo alcune. Per maggiori dettagli circa la definizione di queste classi di grafi si veda [1].

`Path` Genera un cammino (un grafo lineare) P_n con n vertici.

`RandomGraph` Genera un grafo *random* con n vertici ed una probabilità p ($0 \leq p \leq 1$) che ogni spigolo del grafo esista o meno.

`RandomTree` Genera un albero *random* con n vertici (e ovviamente $n-1$ spigoli).

`CompleteBinaryTree` Genera un albero binario completo cono n vertici.

`Cycle` Genera un ciclo C_n con n vertici.

`Star` Genera un grafo "stella" con n vertici, ossia un albero con un vertice di grado $n-1$.

`Wheel` Genera il grafo "ruota" con n vertici, ottenuto come somma del grafo con un solo vertice e del ciclo con $n-1$ vertici.

`CompleteGraph` Genera il grafo completo K_n con n vertici, ossia il grafo in cui esiste uno spigolo che collega ogni coppia di vertici distinti. Se vengono specificati più di un un parametro (es.: $p_1, p_2, \ldots, p_k$) allora viene generato un grafo multipartito completo $K_{p_1,p_2,\ldots,p_k}$, ottenuto come somma di k insiemi stabili (grafi privi di spigoli) di dimensione $p_1, p_2, \ldots, p_k$. Lo stesso grafo può essere ottenuto con la funzione `CompleteKPartiteGraph`.

`GridGraph` Genera un grafo a griglia di dimensione $p_1 \times p_2 \times \cdots \times p_k$; se $k > 3$ allora il grafo non può essere visualizzato.

`LineGraph` Genera il *line graph* del grafo G passato come argomento della funzione; il *line graph* di $G = (V,E)$ è il grafo che ha per vertici gli spigoli di G e in cui due vertici sono adiacenti solo se i rispettivi spigoli in G erano incidenti (avevano un estremo in comune).

`IntervalGraph` Genera il grafo intervallo calcolato su una lista di intervalli chiusi sulla retta reale passata come argomento della funzione; un grafo intervallo ha un vertice per ogni intervallo dell'insieme e due vertici sono adiacenti se e solo se i due intervalli rispettivi si intersecano.

`Harary` Genera un grafo di Harary, ossia il più piccolo grafo k-connesso con n vertici.

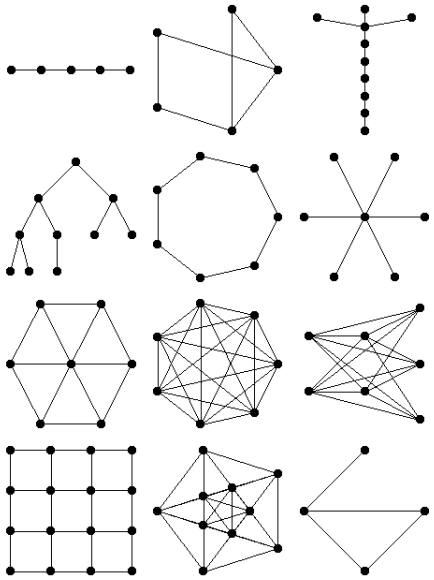
`PetersenGraph` Genera il grafo di Petersen.

`ThomassenGraph` Genera il grafo di Thomassen, ossia un grafo G privo di cicli Hamiltoniani, ma per cui ogni sottografo $G - \{v\}$ contiene un ciclo Hamiltoniano.

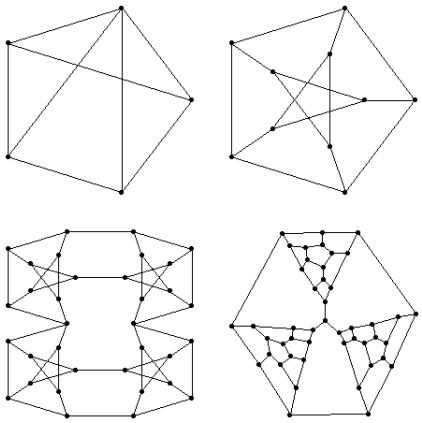
`TutteGraph` Genera il grafo di Tutte, il primo esempio conosciuto di grafo planare 3-connesso, 3-regolare, che non sia Hamiltoniano.

La funzione `ShowGraphArray` consente di visualizzare una "matrice" di grafi, disposti per righe e per colonne. Utilizzando questa funzione sono stati prodotti i grafi riportati nelle figure seguenti, esemplificativi delle funzioni descritte in precedenza:

```
In[1]:= ShowGraphArray[{
          { Path[5], RandomGraph[5,0.4], RandomTree[10] },
          { CompleteBinaryTree[10], Cycle[7], Star[7] },
          { Wheel[7], CompleteGraph[7], CompleteGraph[2,2,3] },
          { GridGraph[4,4], LineGraph[CompleteGraph[5]],
            IntervalGraph[{{1,3}, {2,5}, {4,10}, {4,7}}] } }]
```



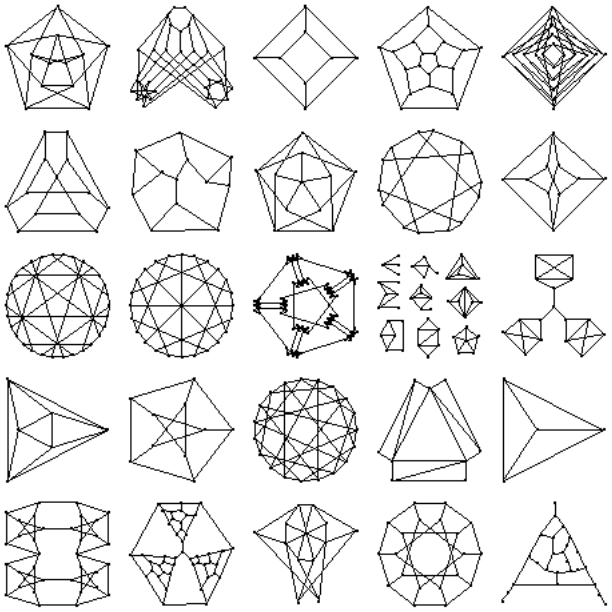
```
Out[1] = -GraphicsArray-
In[2]:= ShowGraphArray[{{Harary[3,5],PetersenGraph}, {ThomassenGraph,TutteGraph}}]
```



```
Out[2] = -GraphicsArray-
```

La funzione `FiniteGraphs` è poco più di una curiosità: consente di produrre una lista di tutti i grafi finiti "interessanti" la cui definizione è presente nel package `Combinatorica`, che possono essere prodotti senza alcun parametro (es.: il grafo di Petersen, il grafo di Thomassen, ecc.). Tale funzione, composta in modo opportuno con la funzione `ShowGraphArray` e con la funzione `Partition` applicata alla lista di grafi, è in grado di produrre un disegno molto interessante riprodotto di seguito:

```
In[1]:= ShowGraphArray[Partition[FiniteGraphs,5,5]];
```



```
Out[2] = -GraphicsArray-
```

5 Operazioni sui grafi

è possibile compiere una serie di operazioni standard sui grafi, in modo da ottenerne altri derivati da questi. Per maggiori informazioni a proposito della definizione e delle proprietà delle operazioni descritte nelle pagine seguenti si veda, ad esempio, [1] e [9].

L'operazione più elementare in questo senso è l'unione fra due o più grafi, ossia il grafo $G = (V,E)$ ottenuto dai grafi $G_1 = (V_1,E_1), G_2 = (V_2,E_2), \ldots, G_k = (V_k,E_k)$ ponendo $V = V_1 \cup V_2 \cup \cdots \cup V_k$ e $E = E_1 \cup E_2 \cup \cdots \cup E_k$. Questa operazione può essere eseguita utilizzando la funzione `GraphUnion`:

```
In[1]:= G = GraphUnion[CompleteGraph[3], Cycle[4]]
Out[1] = -Graph:<7, 7, Undirected>-
```

La somma di due o più grafi con lo stesso numero n di vertici è dato dal multigrafo ottenuto prendendo come vertici l'insieme $\{1,2,\ldots,n\}$ e come insieme degli spigoli, l'unione degli spigoli dei grafi sommati. La funzione `GraphSum` produce la somma tra grafi:

```
In[1]:= G = GraphSum[Cycle[5], Star[5]]
Out[1] = -Graph:<9, 5, Undirected>-
```

L'operazione di *join*, spesso indicata con "$\oplus$", tra due o più grafi consiste nel creare il grafo dato dall'unione dei grafi di partenza insieme con tutti gli spigoli che è possibile creare tra i vertici di grafi distinti. Se $G_1 = (V_1,E_1), G_2 = (V_2,E_2), \ldots, G_p(V_p,E_p)$ sono i grafi di partenza, allora $G = G_1 \oplus G_2 \oplus \cdots \oplus G_p$ è dato dall'insieme dei vertici $V = V_1 \cup V_2 \cup \cdots \cup V_p$ e dall'insieme degli spigoli $E = E_1 \cup E_2 \cup \cdots \cup E_p \cup \{(v_{h_i}, v_{k_j}) : v_{h_i} \in V_h$ e $v_{k_j} \in V_k$ per ogni $h \neq k = 1,\ldots,p$ e $i = 1,\ldots,|V_h|, j = 1,\ldots,|V_k|\}$. La funzione `GraphJoin` implementa questa operazione:

```
In[1]:= G = GraphJoin[Path[3], EmptyGraph[2]]
Out[1] = -Graph:<8, 5, Undirected>-
```

Dato un grafo $G = (V,E)$ il suo complementare $\overline{G} = (V,\overline{E})$ è definito ponendo $\overline{E} = \{(u,v) : u,v \in V(G)$ e $(u,v) \notin E(G)\}$. Dunque il complementare del grafo completo K_n è il grafo vuoto con n vertici privo di spigoli, e viceversa; è interessante notare che il complementare di un P_4 è ancora un P_4. Per ottenere il complementare di un grafo si può utilizzare la funzione `GraphComplement`:

```
 In[1]:= G = GraphComplement[Cycle[5]]
Out[1] = -Graph:<5, 5, Undirected>-
```

Dati due grafi $G_1 = (V,E_1)$ e $G_2 = (V,E_2)$ con lo stesso numero di vertici, $V = \{1,2,\ldots,n\}$, si definisce il grafo differenza $G = G_1 - G_2$ ponendo $G = (V,E)$, dove $E = E_1 - E_2$. Dati $p \geq 1$ grafi $G_1 = (V,E_1), G_2 = (V,E_2), \ldots, G_p = (V,E_p)$ con ugual numero di vertici, $V = \{1,2,\ldots,n\}$, si definisce il grafo intersezione $G = G_1 \cap G_2 \cap \cdots \cap G_p$ ponendo $G = (V,E)$, dove $E = E_1 \cap E_2 \cap \cdots \cap E_p$. Le due funzioni che implementano queste operazioni sono rispettivamente `GraphDifference` e `GraphIntersection`:

```
 In[1]:= G = GraphDifference[CompleteGraph[5], Path[5]]
Out[1] = -Graph:<6, 5, Undirected>-
 In[2]:= GraphIntersection[CompleteGraph[5], Path[5]]
Out[2] = -Graph:<4, 5, Undirected>-
```

Dati p grafi distinti $G_1 = (V_1,E_1), \ldots, G_p = (V_p,E_p)$, si definisce il grafo prodotto $G = G_1 \times \cdots \times G_p$ ponendo $V = V_1 \times \cdots \times V_p$ ed $E = \{(\langle u_{i_1}, ..., u_{i_p}\rangle, \langle v_{i_1}, ..., v_{i_p}\rangle)$ tale che $(u_{i_j}, v_{i_j}) \in E_j$ e $u_{i_k} = v_{i_k}$ per ogni $i_k \neq i_j, i_k, i_j = 1, ..., p\}$. La funzione che consente di costruire il grafo prodotto è `GraphProduct`:

```
 In[1]:= G = GraphProduct[Path[3], Cycle[4]]
Out[1] = -Graph:<20, 12, Undirected>-
```

Dato un grafo $G = (V,E)$ ed un sottoinsieme di vertici $V' \subseteq V$, il sottografo di G indotto da V' è il grafo $G[V'] = (V',E')$, dove $E' = \{(u,v) \in E$ tali che $u,v \in V'\}$. La funzione `InduceSubgraph` restituisce il sottografo di G indotto da un insieme di vertici V:

```
 In[1]:= G = InduceSubgraph[Wheel[5], {1,3,5}]
Out[1] = -Graph:<2, 3, Undirected>-
```

6 Verifica di proprietà e calcolo di invarianti su grafi

Il pacchetto `DiscreteMath::Combinatorica` mette a disposizione un numero veramente elevato di funzioni per la verifica di specifiche proprietà dei grafi; molto spesso si tratta di funzioni che implementano algoritmi ben noti per la risoluzione di problemi di ottimizzazione discreta. A tal proposito per maggiori informazioni sugli algoritmi utilizzati si può fare riferimento a [5] e [2].

Un grafo si dice connesso se per ogni coppia di vertici (u,v) esiste un cammino $p : u \rightsquigarrow v$ che li collega. Le componenti connesse di un grafo $G = (V,E)$ sono i sottografi connessi massimali di G. Se il grafo è connesso allora è costituito da un'unica componente connessa. Le funzioni `ConnectedQ` e `ConnectedComponents` consentono rispettivamente di verificare se un determinato grafo è connesso e di produrre le componenti connesse del grafo; la prima delle due funzioni è di tipo booleano (termina con la lettera "Q") e dunque restituisce valore *vero* o *falso*. Ad esempio:

```
 In[1]:= ConnectedQ[GraphUnion[Cycle[3], Path[2]]]
Out[1] = False
 In[2]:= ConnectedComponents[GraphUnion[Cycle[3], Path[2]]]
Out[2] = {{1, 2, 3}, {4, 5}}
```

La distanza $\delta_G(u,v)$ tra due vertici u e v di un grafo G è data dalla lunghezza del cammino più breve che li collega; se i due vertici non sono raggiungibili l'uno dall'altro allora $\delta_G(u,v) = \infty$. L'eccentricità $\varepsilon_G(v)$ di un vertice del grafo è la massima distanza di tale vertice da ogni altro vertice del grafo. Il diametro di un grafo $G = (V,E)$ è dato dalla lunghezza del più lungo cammino minimo tra tutte le coppie di vertici del grafo: per ogni possibile coppia di vertici $u,v \in V(G)$ si considera la lunghezza del cammino minimo che li unisce e tra tutti i cammini minimi, al variare di $u,v \in V(G)$, si sceglie quello di lunghezza massima. Naturalmente se il grafo non è connesso ha diametro infinito; inoltre in un albero il diametro coincide con la profondità dell'albero stesso. Il diametro è dunque la massima eccentricità dei vertici del grafo. La funzione `Diameter` restituisce il diametro del grafo G. Il centro del grafo è costituito dal vertice di eccentricità minima (un grafo può avere più centri); l'eccentricità di tale vertice è il raggio del grafo. La funzione `Radius` restituisce il raggio di G, mentre `GraphCenter` restituisce il centro (uno o più vertici).

```
 In[1]:= Diameter[CompleteBinaryTree[10]]
Out[1] = 5
 In[2]:= Radius[CompleteBinaryTree[10]]
Out[2] = 3
 In[1]:= GraphCenter[CompleteBinaryTree[10]]
Out[1] = {1, 2}
```

Un punto di articolazione di un grafo G è un vertice che, se rimosso, sconnette il grafo; analogamente un *bridge* è uno spigolo che, se rimosso, sconnette il grafo. Per ottenere gli eventuali punti di articolazione e i bridge di un grafo si possono utilizzare rispettivamente le funzioni `ArticulationVertices` e `Bridges`:

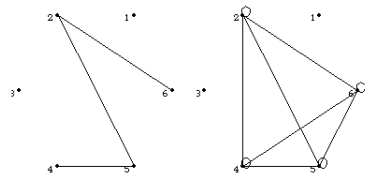
```
 In[1]:= ArticulationVertices[Star[5]]
Out[2] = {5}
 In[2]:= ArticulationVertices[CompleteGraph[5]]
Out[2] = {}
 In[3]:= Bridges[Star[5]]
Out[3] = {{2, 5}, {3, 5}, {4, 5}, {1, 5}}
```

Le funzioni `EdgeConnectivity` e `VertexConnectivity` restituiscono rispettivamente il minimo numero di spigoli e di vertici che devono essere rimossi dal grafo per sconnetterlo:

```
 In[1]:= EdgeConnectivity[Wheel[5]]
Out[2] = 3
 In[2]:= VertexConnectivity[Wheel[5]]
Out[2] = 3
```

La chiusura transitiva di un grafo $G = (V,E)$ è un supergrafo $G' = (V,E')$ di G ottenuto ponendo $E' = \{(u,v) : u,v \in V \text{ e } \exists p : u \rightsquigarrow v \text{ in } G\}$. Da notare che E' contiene anche tutti i cappi (cicli di lunghezza 1) sui vertici di G. `TransitiveClosure` restituisce la chiusura transitiva di un grafo G.

```
 In[1]:= G = RandomGraph[6, 0.4];
 In[2]:= ShowGraphArray[{G,TransitiveClosure[G]},VertexNumber->On]
```



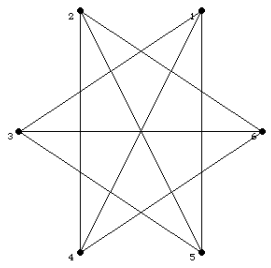
```
Out[2] = -GraphicsArray-
```

Due grafi G e H si dicono isomorfi se è possibile creare una corrispondenza biunivoca φ tra $V(G)$ e $V(H)$ che mandi vertici adiacenti in vertici adiacenti, ossia tale che $(u,v) \in E(G) \iff (\varphi(u),\varphi(v)) \in E(H) \forall u,v \in V(G)$. Dunque di fatto due grafi G e H sono isomorfi se il disegno di G può essere trasformato facendolo coincidere con quello di H senza alterare i collegamenti (gli spigoli) esistenti tra i vertici del grafo stesso. La funzione `Isomorphism` consente di costruire, se esiste, un isomorfismo tra due grafi, ossia una permutazione dell'insieme dei vertici che consente di trasformare il primo grafo nel secondo. L'opzione `All` permette di esibire tutti gli isomorfismi esistenti tra i due grafi. Se viene specificato un solo grafo, allora vengono elencati tutti gli automorfismi di tale grafo (lo stesso risultato lo si ottiene anche con la funzione `Automorphisms`). La funzione booleana `IsomorphicQ` restituisce il valore *vero* se i due grafi specificati come argomento sono isomorfi, altrimenti restituisce il valore *falso*.

```
 In[1]:= IsomorphicQ[Path[3],
           GraphJoin[EmptyGraph[2], EmptyGraph[1]]]
Out[1] = True
 In[2]:= Isomorphism[Path[3],
           GraphJoin[EmptyGraph[2], EmptyGraph[1]]]
Out[2] = {1, 3, 2}
 In[3]:= Isomorphism[Path[3],
           GraphJoin[EmptyGraph[2], EmptyGraph[1]], All]
Out[3] = {{1, 3, 2}, {2, 3, 1}}
 In[4]:= Isomorphism[Path[3]]
Out[4] = {{1, 2, 3}, {3, 2, 1}}
```

Un ciclo in un grafo G è un cammino che inizia e termina nello stesso vertice. Il ciclo è semplice se non contiene propriamente altri cicli. La funzione `FindCycle` restituisce un ciclo semplice del grafo, mentre con la funzione `ExtractCycles` si ottiene un insieme massimale di cicli semplici disgiunti (ossia senza nessuno spigolo in comune) presenti nel grafo; non fornisce tutti i cicli semplici contenuti nel grafo, a meno che questi non siano tutti disgiunti. La funzione `DeleteCycle`, infine, elimina dal grafo gli spigoli che formano il ciclo specificato.

```
 In[1]:= FindCycle[CompleteGraph[6]]
Out[1] = {3, 1, 2, 3}
 In[2]:= ExtractCycles[CompleteGraph[6]]
Out[2] = {{5, 3, 4, 5}, {5, 1, 4, 2, 5}, {3, 1, 2, 3}}
 In[3]:= ShowGraph[DeleteCycle[CompleteGraph[6], {1, 2, 3, 4, 5, 6, 1}],
           VertexNumber -> On]
```




```
Out[3] = -Graphics-
```

La funzione booleana `AcyclicQ` restituisce valore *vero* se il grafo è aciclico (è privo di cicli), *falso* altrimenti. Un albero, come è ben noto, è un grafo connesso e aciclico; la funzione `TreeQ` consente di verificare se il grafo passato come argomento è un albero. Dunque `TreeQ[G]` è equivalente a `AcyclicQ[G] && ConnectedQ[G]`.

Si dice che un grafo è Euleriano se ammette un cammino (anche non semplice) che consente di percorrere ogni spigolo del grafo esattamente una volta. La funzione booleana `EulerianQ` consente di verificare questa proprietà su un grafo G, mentre `EulerianCycle` restituisce, se esiste, un ciclo Euleriano presente nel grafo. Un grafo è Hamiltoniano se contiene un ciclo semplice che passa per ogni vertice. La funzione `HamiltonianCycle` restituisce un ciclo Hamiltoniano, se esiste; con l'opzione `All` visualizza tutti i cicli Hamiltoniani presenti nel grafo.

```
 In[1]:= EulerianQ[CompleteGraph[5]]
Out[1] = True
 In[2]:= EulerianCycle[CompleteGraph[5]]
Out[2] = {2, 3, 1, 4, 5, 3, 4, 2, 5, 1, 2}
 In[3]:= HamiltonianCycle[CompleteGraph[3,3]]
Out[3] = {1, 4, 2, 5, 3, 6, 1}
 In[4]:= HamiltonianCycle[Complete[3,3], All]
Out[4] = {{1, 4, 2, 5, 3, 6, 1}, {1, 4, 2, 6, 3, 5, 1},
          {1, 4, 3, 5, 2, 6, 1}, {1, 4, 3, 6, 2, 5, 1},
          {1, 5, 2, 4, 3, 6, 1}, {1, 5, 2, 6, 3, 4, 1},
          {1, 5, 3, 4, 2, 6, 1}, {1, 5, 3, 6, 2, 4, 1},
          {1, 6, 2, 4, 3, 5, 1}, {1, 6, 2, 5, 3, 4, 1},
          {1, 6, 3, 4, 2, 5, 1}, {1, 6, 3, 5, 2, 4, 1}}
```

Per rappresentare con un grafo il modello di un contesto reale è spesso utile attribuire un peso agli spigoli o ai vertici del grafo. È possibile farlo in modo esplicito con le funzioni `SetEdgeWeights` e `SetVertexWeights`, fornendo come argomento delle funzioni, oltre al grafo, rispettivamente anche una lista di spigoli o una lista di vertici e la lista dei pesi che si intende assegnare a quegli spigoli o a quei vertici; ai vertici e agli spigoli di ogni grafo di default viene assegnato rispettivamente il peso 0 e il peso 1. Per visualizzare i pesi associati agli spigoli si deve utilizzare la funzione `Edges` con l'opzione `EdgeWeight`, oppure, più semplicemente, la funzione `GetEdgeWeights`; analogamente, per visualizzare i pesi associati ai vertici si deve utilizzare la funzione `Vertices` con l'opzione `All`, oppure la funzione `GetVertexWeights`. Ad esempio:

```
 In[1]:= G = CompleteGraph[4];
 In[2]:= G = SetEdgeWeights[G, {{1,2}, {1,3}, {2,3}}, {1,5,7}];
 In[3]:= G = SetVertexWeights[G, {1,2,3}, {2,4,6}];
 In[4]:= Edges[G, EdgeWeight]
Out[4] = {{1, 2, 1}, {{1, 3}, 5}, {{1, 4}, 1}, {{2, 3}, 7},
          {{2, 4}, 1}, {{3, 4}, 1}}
 In[5]:= GetVertexWeights[G]
Out[5] = {2, 4, 6, 0}
```

Se alle due funzioni `SetEdgeWeights` e `SetVertexWeights` viene passato come unico argomento il grafo, allora i pesi saranno assegnati in modo casuale selezionandoli nell'intervallo $[0,1]$.

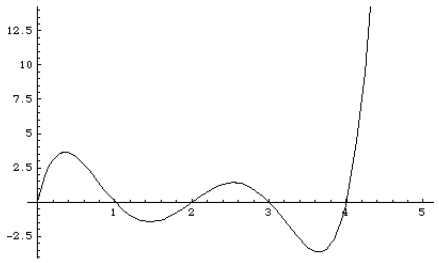
La funzione `CostOfPath` restituisce il costo del cammino p sul grafo pesato G, inteso come la somma dei pesi associati agli spigoli di p. Se il grafo non è pesato la funzione restituisce quindi la lunghezza del cammino. Se il cammino p non è definito su G la funzione restituisce ∞.

```
 In[1]:= CostOfPath[CompleteBinaryTree[10], {1, 2, 5, 10}]
Out[1] = 3
 In[2]:= CostOfPath[CompleteBinaryTree[10], {1, 2, 3, 4}]
Out[2] = ∞
```

Un problema molto interessante è quello della colorazione di un grafo G: si tratta di calcolare quale è il minimo numero di colori necessari a colorare i vertici del grafo in modo tale che due vertici adiacenti non abbiano mai lo stesso colore. È possibile associare ad ogni grafo un polinomio $P(x)$,

detto polinomio cromatico, che indica in quanti modi differenti è possibile colorare i vertici di G utilizzando x colori. La funzione `ChromaticPolinomial` restituisce il polinomio cromatico di G nella variabile x. Il numero cromatico di un grafo G, indicato tradizionalmente con $\chi(G)$, rappresenta il minimo numero di colori necessari per colorare G. La funzione `ChromaticNumber` restituisce il valore di $\chi(G)$. Naturalmente risulta che $P(n) = 0$ per ogni naturale $n < \chi(G)$. Con la funzione `VertexColoring` si ottiene una possibile colorazione approssimata (non è necessariamente ottimale) del grafo G, calcolata utilizzando l'euristica di Brelaz[3].

```
 In[1]:= ChromaticNumber[CompleteGraph[5]]
Out[1] = 5
 In[2]:= ChromaticPolynomial[CompleteGraph[5], x]
Out[2] = (-4 + x) (-3 + x) (-2 + x) (-1 + x) x
 In[3]:= P[x_] := ChromaticPolynomial[CompleteGraph[5], x]
 In[4]:= Plot[P[x], {x, 0, 5}]
```



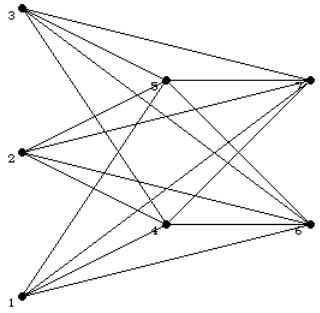

```
Out[4] = -Graphics-
 In[5]:= P[4]
Out[5] = 0
 In[6]:= P[5]
Out[6] = 120
 In[7]:= VertexColoring[CompleteGraph[5]]
Out[7] = {1, 2, 3, 4, 5}
```

Una clique di un grafo G è un suo sottografo completo massimale, ossia un sottografo completo di G che non sia contenuto propriamente in nessun altro sottografo completo. La funzione `CliqueQ` verifica se un certo sottoinsieme di vertici costituisce una clique per il grafo G. La funzione `MaximumClique` restituisce una clique di G di cardinalità massima. Viceversa un insieme indipendente è un insieme massimale di vertici che non siano a due a due adiacenti. La funzione `IndependentSetQ` verifica se un certo sottoinsieme di vertici di G è indipendente, mentre `MaximumIndependentSet` restituisce un insieme indipendente di G di cardinalità massima. Una copertura di vertici di $G = (V,E)$ è un insieme $V' \subseteq V$ di vertici tali che ogni spigolo $e \in E$ abbia almeno un estremo in V'. La funzione `VertexCoverQ` verifica se un certo insieme di vertici è una copertura per il grafo G, mentre la funzione `MinimumVertexCover` restituisce una copertura di vertici di cardinalità minima.

```
 In[1]:= G = CompleteGraph[3,2,2];
 In[2]:= ShowGraph[G, VertexNumber->On]

Out[2] = -Graphics-
 In[3]:= CliqueQ[G, {3,5,7}]
Out[3] = True
 In[4]:= MaximumClique[G]
Out[4] = {1, 4, 6}
```

[3]D. Brelaz, *New methods to color the vertices of a graph*, Communications of the ACM, vol. 22, 251-256, 1979.




```
 In[5]:= IndependentSetQ[G, {1,2,3}]
Out[5] = True
 In[6]:= MaximumIndependentSet[G]
Out[6] = {1, 2, 3}
 In[7]:= VertexCoverQ[G, {1,2,3,4,5}]
Out[7] = True
 In[8]:= MinimumVertexCover[G]
Out[8] = {4, 5, 6, 7}
```

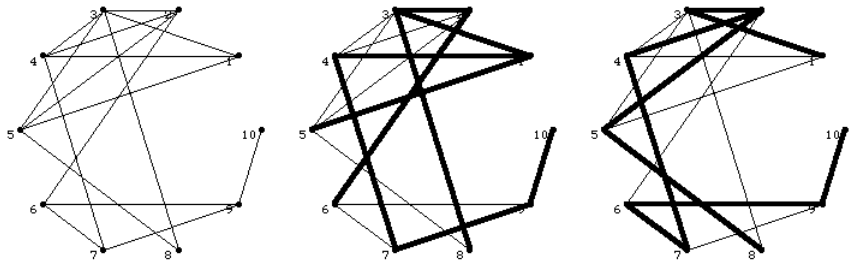
7 Altri algoritmi fondamentali sui grafi

Esistono innumerevoli algoritmi interessanti per la soluzione di problemi di ottimizzazione su grafi, alcuni dei quali sono implementati dalle funzioni viste nelle pagine precedenti per il calcolo di determinate proprietà o di invarianti sui grafi. Di seguito riportiamo una breve descrizione delle funzioni che implementano alcuni algoritmi particolarmente noti.

La visita di un grafo consiste nella costruzione di un albero (o di una foresta di alberi disgiunti) che contengano tutti i cammini che, senza effettuare cicli, consentono di raggiungere ogni vertice del grafo da un vertice definito come sorgente della visita. Naturalmente cambiando sorgente della visita possono essere costruiti alberi di visita differenti; inoltre, a partire da un medesimo vertice scelto come sorgente, è possibile individuare anche più di un albero di visita. Esistono due algoritmi fondamentali molto noti ed efficienti per la visita di un grafo: l'algoritmo BFS (*breadth first search*) per la visita in ampiezza e l'algoritmo DFS (*depth first search*) per la visita in profondità. Le funzioni `BreadthFirstTraversal` e `DepthFirstTraversal` forniscono una implementazione di tali algoritmi e, con l'opzione `Edge`, restituiscono la lista di spigoli che compongono rispettivamente l'albero (o la foresta) di visita in ampiezza e in profondità:

```
 In[1]:= G = RandomGraph[10, 0.3];
 In[2]:= BreadthFirstTraversal[G, 1]
Out[2] = {1, 5, 2, 3, 4, 8, 7, 6, 9, 10}
 In[3]:= BreadthFirstTraversal[G, 1, Edge]
Out[3] = {{1,3},{1,4},{1,5},{3,2},{3,8},{4,7},{2,6},{7,9},{9,10}}
 In[4]:= DepthFirstTraversal[G, 1, Edge]
Out[4] = {{1,3},{3,2},{2,4},{4,7},{7,6},{6,9},{9,10},{2,5},{5,8}}
 In[5]:= ShowGraphArray[{G,
             Highlight[G,{BreadthFirstTraversal[G, 1, Edge]}],
             Highlight[G,{DepthFirstTraversal[G, 1, Edge]}]}, VertexNumber -> On]

Out[5] = -GraphicsArray-
```



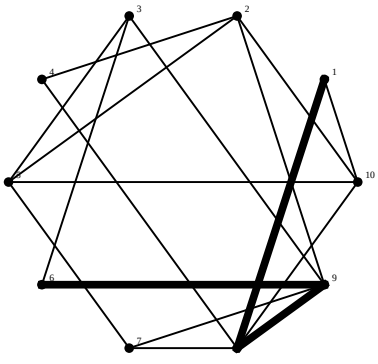
Dato un grafo $G = (V,E)$, uno *spanning tree*, o albero ricoprente, di G è un sottografo $T = (V,E')$ aciclico e connesso di G, ossia è un albero costruito su G utilizzando tutti i suoi vertici e solo alcuni dei suoi spigoli (esattamente $|V|-1$ spigoli, in modo da garantire sia la connessione che l'aciclicità di T).

Un cammino minimo dal vertice u al vertice v sul grafo $G = (V,E)$ è un cammino che collega i due vertici con il minor numero di spigoli. La funzione `ShortestPath[G,u,v]` restituisce una successione di k vertici distinti che identifica il cammino minimo tra u e v. Ad esempio:

```
 In[1]:= G = RandomGraph[10,0.4]
Out[1] = - Graph: <17, 10, Undirected> -
 In[2]:= ShortestPath[G, 1, 6]
Out[2] = {1, 8, 9, 6}
 In[3]:= Partition[ShortestPath[G, 1, 6], 2, 1]
Out[3] = {{1, 8}, {8, 9}, {9, 6}}
```

La distanza tra due vertici $u,v \in V(G)$ è data dalla lunghezza del cammino minimi da u a v; utilizzando Mathematica, proseguendo l'esempio precedente, potremmo quindi definire la funzione `Distanza`:

```
 In[4]:= Distanza[g_,u_,v_]:=Length[Partition[ShortestPath[g,u,v],2,1]]
 In[5]:= Distanza[G, 1, 6]
Out[5] = 3
In[6]:= ShowGraph[Highlight[G, {Partition[ShortestPath[G,1,6], 2, 1]}],
              VertexLabel->True]
```

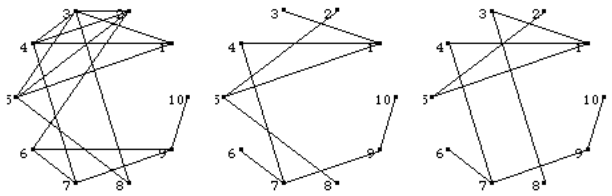


È possibile visualizzare una matrice con tutte le distanze esistenti tra tutti vertici del grafo (distanza tra ogni coppia di vertici):

```
 In[7]:= TableForm[Table[Distanza[G, a, b],{a,1,V[G]},{b,1,V[G]}]]
Out[7]//TableForm=
  0   2   3   2   2   3   2   1   2   1
  2   0   2   1   1   2   2   2   1   1
  3   2   0   3   1   1   2   2   1   2
  2   1   3   0   2   3   2   1   2   2
  2   1   1   2   0   2   1   2   2   1
  3   2   1   3   2   0   2   2   1   3
  2   2   2   2   1   2   0   1   1   2
  1   2   2   1   2   2   1   0   1   1
  2   1   1   2   2   1   1   1   0   2
  1   1   2   2   1   3   2   1   2   0
```

Un albero ricoprente di cammini minimi di G con radice in $v \in V(G)$ è uno spanning tree costruito in modo tale che i cammini minimi da v ad ogni altro vertice del grafo G siano i cammini in T. La funzione `ShortestPathSpanningTree` costruisce tale albero ricoprente su G, utilizzando come radice il vertice v. In generale tutti gli spigoli sono considerati di lunghezza (o peso) unitario, tuttavia è possibile applicare la funzione anche a grafi con pesi associati agli spigoli: in tal caso la lunghezza di un cammino sarà data dalla somma dei pesi dei suoi spigoli. Per il calcolo dell'albero ricoprente vengono utilizzati gli algoritmi di Dijkstra o di Bellman-Ford; è possibile indicare quale dei due algoritmi si desidera utilizzare specificando l'opzione `Algorithm` che può assumere i valori `Dijkstra` o `BellmanFord`.

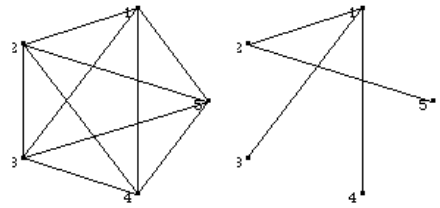
```
 In[1]:= G = RandomGraph[10, 0.3];
 In[2]:= ShowGraphArray[{G,
           ShortestPathSpanningTree[G,1,Algorithm->Dijkstra],
           ShortestPathSpanningTree[G,1,Algorithm->BellmanFord]},
           VertexNumber -> On]
```



```
Out[2] = -GraphicsArray-
```

Su un grafo con pesi associati agli spigoli è possibile risolvere un'istanza del problema del commesso viaggiatore (TSP: *traveling salesman problem*), ossia della ricerca di un ciclo Hamiltoniano di peso minimo, e il problema della costruzione dell'albero ricoprente di peso minimo (*minimum spanning tree*). La funzione `TravelingSalesman` implementa l'algoritmo per la costruzione del ciclo che risolve il primo problema (se tale ciclo esiste), mentre `MinimumSpanningTree` restituisce l'albero ricoprente con il peso minimo.

```
 In[1]:= G = CompleteGraph[5];
 In[2]:= G = SetEdgeWeights[G];
 In[3]:= GetEdgeWeights[G]
Out[3] = {0.399651, 0.178447, 0.442441, 0.791806, 0.997585,
          0.669053, 0.0951515, 0.608909, 0.591402, 0.634163}
 In[4]:= TravelingSalesman[G]
Out[4] = {1, 3, 4, 2, 5, 1}
 In[5]:= ShowGraphArray[{G, MinimumSpanningTree[G]}]
```




```
Out[5] = -GraphicsArray-
```

Se il grafo G non è pesato (di default agli spigoli del grafo viene assegnato il peso unitario), la funzione `MinimumSpanningTree` restituisce un albero ricoprente qualsiasi.

Un *matching* su un grafo $G = (V,E)$ è un insieme di spigoli $E' \subseteq E$ a due a due non incidenti (senza estremi in comune). La funzione `MaximalMatching` restituisce un matching massimale su G. Il problema del matrimonio stabile (*stable marriage problem*) è un problema di ottimizzazione combinatoria molto famoso: dati due insiemi U e D disgiunti (ad esempio un insieme di uomini e un insieme di donne), e due liste di preferenza di ogni elemento di un insieme rispetto agli elementi dell'altro (la lista delle preferenze di ognuno degli uomini rispetto alle donne e viceversa), un matching tra i due insiemi è stabile se non contiene nessuna coppia di elementi $(u,d) \in U \times D$ tale che sia u che d si preferiscono rispetto agli elementi a cui sono stati abbinati nel matching. La funzione `StableMarriage` risolve il problema trovando un matching stabile tra i due insiemi di uguale cardinalità; alla funzione si devono passare come argomento le liste di preferenza di ciasun elemento dei due insiemi.

```
 In[1]:= MaximalMatching[CompleteGraph[9]]
Out[1] = {{1, 2}, {3, 4}, {5, 6}, {7, 8}}
 In[2]:= StableMarriage[{{1,3,2},{1,3,2},{3,2,1}},
           {{1,3,2},{1,2,3},{3,2,1}}]
Out[2] = {1, 2, 3}
```

8 Elenco alfabetico delle funzioni

Di seguito riportiamo una descrizione sintetica che riepiloga lo scopo delle funzioni descritte nelle pagine precedenti.

`%` Restituisce il risultato dell'ultima elaborazione effettuata.

`AcyclicQ[`G**`]`** Verifica se il grafo G è aciclico (booleana).

`AddEdge[`G**`,`**e**`]`** Restituisce il grafo ottenuto aggiungendo a G lo spigolo e.

`AddEdges[`G**`,{`**e_1**`,...,`**e_k**`}]`** Restituisce il grafo ottenuto aggiungendo a G gli spigoli $e_1,\ldots,e_k$.

`AddVertex[`G**`]`** Restituisce il grafo ottenuto aggiungendo un vertice a G.

`AddVertices[`G**`,`**n**`]`** Restituisce il grafo ottenuto aggiungendo a G altri n vertici.

`ArticulationVertices[`G**`]`** Restituisce i punti di articolazione del grafo G.

`Automorphism[`G**`]`** Restituisce un automorfismo su G (una permutazione dei vertici) se esiste.

`BreadthFirstTraversal[`G**`,`**v**`]`** Restituisce la sequenza di vertici incontrati nella visita in ampiezza del grafo G a partire dal vertice v.

Bridge[G] Restituisce gli spigoli *bridge* di G.

ChromaticPolynomial[G,x] Restituisce il polinomio cromatico di G nella variabile x.

ChromaticNumber[G] Restituisce il numero cromatico $\chi(G)$ del grafo G.

CliqueQ[G,{v_1,...,v_k}] Verifica se l'insieme di vertici $\{v_1,\ldots,v_k\}$ è una clique del grafo G (booleana).

Complement[A,B_1,...,B_n] Restituisce l'insieme $C = A - (B_1 \cup \cdots \cup B_n)$.

CompleteBinaryTree[n] Restituisce un albero binario completo con n vertici.

CompleteGraph[n] Restituisce il grafo completo con n vertici K_n.

Compositions[n,k] Restituisce tutte gli insiemi di naturali di k elementi la cui somma sia n.

ConnectedComponents[G] Restituisce le componenti connesse del grafo G.

ConnectedQ[G] Verifica se il grafo G è connesso (booleana).

Cos[x] Restituisce $cos(x)$.

CostOfPath[G,p] Restituisce il costo del cammino p sul grafo G.

Cycle[n] Restituisce il grafo C_n composto da un ciclo su n vertici.

DeleteCycle[G,C] Elimina dal grafo G gli spigoli che compongono il ciclo C; restituisce un grafo.

DeleteVertices[G,{v_1,...,v_k}] Restituisce il grafo ottenuto da G eliminando i vertici $v_1,\ldots,v_k$ e gli spigoli loro incidenti.

DepthFirstTraversal[G,v] Restituisce la sequenza di vertici incontrati nella visita in profondità del grafo G a partire dal vertice v.

EdgeConnectivity[G] Restituisce il minimo numero di spigoli che devono essere rimossi da G per sconnettere il grafo.

Edges[G] Restituisce gli spigoli del grafo G; con l'opzione `EdgeWeight` restituisce anche il peso associato ad ogni spigolo.

EmptyGraph[n] Restituisce un grafo con n vertici, privo di spigoli.

EulerianCycle[G] Restituisce un ciclo Euleriano in G.

EulerianQ[G] Verifica se il grafo G è Euleriano (booleana).

ExtractCycles[G] Restituisce un insieme di cicli semplici privi di spigoli in comune ini G.

FindCycle[G] Restituisce un ciclo semplice su G.

FromAdjacencyLists[{l_1,...,l_n}] Restituisce il grafo costruito sulla base delle liste di adiacenza $l_1,\ldots,l_n$.

FromAdjacencyMatrix[M] Restituisce il grafo costruito in base alla matrice di adiacenza M.

GetEdgeWeights[G] Restituisce la lista dei pesi associati agli spigoli del grafo.

GetVertexWeights[G] Restituisce la lista dei pesi associati ai vertici del grafo.

GraphCenter[G] Restituisce il centro (o i centri) del grafo G.

GraphComplement[G] Restituisce il grafo $\overline{G}$ complementare di G.

GraphDifference[G_1, G_2] Restituisce il grafo $G = G_1 - G_2$.

GraphIntersection[$G_1, \dots, G_n$] Restituisce il grafo $G = G_1 \cap \cdots \cap G_n$.

GraphJoin[$G_1, \dots, G_n$] Restituisce un grafo ottenuto eseguendo il *join* tra i grafi $G_1, \dots, G_n$.

GraphProduct[$G_1, \dots, G_n$] Restituisce il grafo $G = G_1 \times \cdots \times G_n$.

GraphSum[$G_1, \dots, G_n$] Restituisce il grafo $G = G_1 \oplus \cdots \oplus G_n$.

GraphUnion[$G_1, \dots, G_n$] Restituisce il grafo $G = G_1 \cup \cdots \cup G_n$.

GridGraph[$p_1, \dots, p_k$] Genera un grafo griglia di dimensione $p_1 \times \cdots \times p_k$.

HamiltonianCycle[G] Restituisce un ciclo Hamiltoniano su G; con l'opzione `All` li restituisce tutti.

Harary[k, n] Genera un grafo di Harary con n vertici.

Highlight[G, {$s_1, \dots, s_k$}] Restituisce il grafo ottenuto da G evidenziando la sequenza $\{s_1, \dots, s_k\}$ di vertici o di spigoli.

IndependentSet[G, {$v_1, \dots, v_k$}] Verifica se l'insieme di vertici $\{v_1, \dots, v_k\}$ è indipendente nel grafo G.

InduceSubgraph[G, {$v_1, \dots, v_k$}] Restituisce il sottografo di G indotto dall'insieme di vertici $\{v_1, \dots, v_k\}$.

Intersection[$A_1, \dots, A_n$] Restituisce l'insieme $A = A_1 \cap \cdots \cap A_n$.

IntervalGraph[{$A_1, \dots, A_k$}] Genera il grafo intervallo calcolato sulla collezione di intervalli chiusi $A_1, \dots, A_k$.

IsomorphicQ[G_1, G_2] Verifica se i grafi G_1 e G_2 sono isomorfi (booleana).

Isomorphism[G_1, G_2] Restituisce un isomorfismo (una permutazione di $V(G_1)$) tra i grafi G_1 e G_2 (se sono isomorfi).

KSetPartitions[A, k] Restituisce tutte le partizioni di A in k sottoinsiemi.

KSubsets[A, k] Restituisce tutti i sottoinsiemi di A con k elementi.

Length[A] Restituisce il numero di elementi dell'insieme A.

LineGraph[G] Genera il *line graph* di G.

M[G] Restituisce il numero di spigoli del grafo G, $m = |E(G)|$.

MatrixForm[M] Visualizza in forma matriciale la tabella M.

MaximalMatching[G] Restituisce un matching massimale sul grafo G.

MaximumClique[G] Restituisce una clique di cardinalità massima di G.

MaximumIndependentSet[G] Restituisce un insieme indipendente di G di cardinalità massima.

MinimumSpanningTree[G] Restituisce l'albero ricoprente di peso minimo sul grafo pesato G.

MinimumVertexCover[G] Restituisce una copertura di vertici di G di cardinalità minima.

N[*expr*,n] Restituisce una rappresentazione numerica approssimata di *expr* con n cifre decimali.

Part[M,i_1,...,i_n] Restituisce l'elemento $M_{i_1,\ldots,i_n}$ della matrice (o del vettore) M.

Partitions[n] Restituisce tutte le partizioni dell'intero n.

Path[n] Restituisce il cammino P_n con n vertici.

PermutationQ[P] Verifica se l'insieme P rappresenta una permutazione di $\{1,\ldots,n\}$ (booleana).

Permutations[A] Restituisce la famiglia di tutte le permutazioni degli elementi dell'insieme A.

Permute[A,P] Restituisce gli elementi dell'insieme A permutati in base alla permutazione P.

PetersenGraph Restituisce il grafo di Petersen.

Radius[G] Restituisce il raggio del grafo G.

RandomGraph[n] Restituisce un grafo random con n vertici.

RandomPermutation[n] Restituisce una permutazione random degli elementi dell'insieme $\{1,\ldots,n\}$.

RandomSubset[A] Restituisce un sottoinsieme di A scelto in modo casuale.

RandomTree[n] Restituisce un albero random con n vertici.

SetEdgeWeights[G,{e_1,...,e_k},{w_1,...,w_k}] Restituisce il grafo pesato ottenuto da G asegnando agli spigoli $e_1,\ldots,e_k$ i pesi $w_1,\ldots,w_k$.

SetVertexWeights[G,{v_1,...,v_k},{w_1,...,w_k}] Restituisce il grafo pesato ottenuto da G asegnando ai vertici $v_1,\ldots,v_k$ i pesi $w_1,\ldots,w_k$.

SetGraphOptions[G,opt_1,...,opt_k] Restituisce il grafo ottenuto da G impostando le opzioni di visualizzazione specificate.

SetPartitions[A] Restituisce tutte le partizioni dell'insieme A. Se l'argomento è un intero n allora restituisce tutte le partizioni di $\{1,\ldots,n\}$.

ShortestPathSpanningTree[G,v] Restituisce l'albero ricoprente di cammini minimi del grafo G con radice in v.

ShowGraph[G**]** Visualizza il grafo G.

ShowGraphArray[{G_1**, ... ,** G_n**}]** Visualizza i grafi $G_1, \ldots, G_n$ disponendoli uno accanto all'altro in una tabella.

Sin[x**]** Restituisce $\sin(x)$.

Sort[A**]** Restituisce gli elementi di A in ordine crescente.

Sqrt[x**]** Per $x \geq 0$ restituisce $\sqrt{x}$.

StableMarriage[{p_1**, ... ,** p_k**}, {**p'_1**, ... ,** p'_k**}]** Restituisce un matching stabile sulla base delle preferenze indicate nelle due liste.

Star[n**]** Restituisce il grafo stella con n vertici (un albero con un vertice di grado $n-1$).

Subsets[A**]** Restituisce la famiglia di tutti i sottoinsiemi di A.

ThomassenGraph Restituisce il grafo di Thomassen.

ToAdjacencyLists[G**]** Restituisce le liste di adiacenza di G.

ToAdjacencyMatrix[G**]** Restituisce la matrice di adiacenza di G.

TransitiveClosure[G**]** Restituisce la chiusura transitiva del grafo G.

TravelingSalesman[G**]** Restituisce (se esiste) un ciclo sul grafo pesato G che risolve il problema del commesso viaggiatore.

TreeQ[G**]** Verifica se il grafo G è un albero (booleana).

TutteGraph Restituisce il grafo di Tutte.

Union[A_1**, ... ,** A_n**]** Restituisce l'insieme $A = A_1 \cup \cdots \cup A_n$.

V[G**]** Restituisce il numero di vertici del grafo G, $|V(G)|$.

VertexColoring[G**]** Restituisce una colorazione approssimata dei vertici del grafo G.

VertexConnectivity[G**]** Restituisce il minimo numero di vertici che devono essere rimossi da G per sconnettere il grafo.

VertexCoverQ[G**, {**v_1**, ... ,** v_k**}]** Verifica se l'insieme di vertici $\{v_1, \ldots, v_k\}$ è una copertura di vertici per G (booleana).

Vertices[G**]** Restituisce le coordinate dei vertici del grafo G nel piano cartesiano; con l'opzione `All` visualizza anche i pesi associati ai vertici.

Wheel[n**]** Restituisce il grafo *wheel* con n vertici.

Riferimenti bibliografici

[1] Andreas Brandstädt, Van Bang Le, Jeremy P. Spinrad, *Graph Classes – A survey*, SIAM Monographs on Discrete Mathematics and Applications, 1999.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduzione agli algoritmi e strutture dati*, terza edizione, McGraw-Hill, 2010.

[3] Pierluigi Crescenzi, Giorgio Gambosi, Roberto Grossi, *Strutture di dati e algoritmi*, Pearson–Addison Wesley, 2006.

[4] Ottavio D'Antona, *Introduzione alla Matematica Discreta*, Apogeo, 1999.

[5] Alan Gibbons, *Algorithmic graph theory*, Cambridge University Press, 1994.

[6] Peter Gritzmann, René Brandenberg, *Alla ricerca della via più breve. Un'avventura matematica*, Springer, 2005.

[7] Giulia Maria Piacentini Cattaneo, *Algebra, un approccio algoritmico*, Decibel – Zanichelli, 2001.

[8] Sriram Pemmaraju, Stephen Skiena, *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*, Cambridge University Press, 2003.

[9] Richard J. Trudeau, *Introduction to Graph Theory*, Dover Publications Inc., New York, 1993.