Lezione del 04-06-2014 e del 05-06-2014.

Esempi di grafi sono le mappe (le città sono i nodi e a ogni arco è associato un peso) e internet.

Cose importanti da ricordarsi: definizione di grafo ordinato e non, grafo semplice, completo, grado di un nodo (numero di archi incidenti su un determinato vertice, non importa se ordinato o no).

Attraversamento dei grafi

Introduzione

Def. termine ***visita di un grafo*** s'intende l'insieme di tutte le operazioni tali per cui partendo da un vertice iniziale si riesce a visitare tutti gli altri nodi passando solo attraverso gli archi che connettono il grafo (se orientato tenendo anche conto del verso). In genere non c'è un vertice "sorgente" (a differenza degli alberi dove la radice era sempre identificata) ma lo decidiamo noi a seconda dell'applicazione.

Gli algoritmi di visita sono divisi in due grandi classi: in ampiezza e in profondità.

Visita in ampiezza

Partendo da un nodo s di partenza, inizio a visitare tutti i nodi a lui adiacenti, poi visito tutti quelli distanti 2, poi quelli distanti 3, e iterando così fino a quando non si visitano tutti quanti.

Partendo da un vertice s di partenza, assegno a ogni nodo un numero chiamato **livello** pari al *numero minimo di archi necessario per collegare la sorgente con l'i-esimo nodo* (il numero minimo di step che devo fare per raggiungere i partendo da s). Prima visito tutti i vertici con livello 1, poi tutti quelli di livello 2, etc. Quando vado a etichettare tutti i nodi devo essere sicuro del livello assegnato.

Con un foglio di carta è semplice farlo, fare un algoritmo efficiente diventa già più complicato.

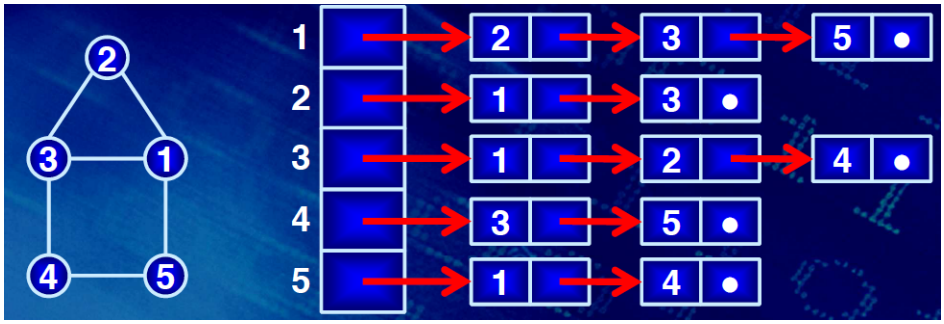
A livello implementativo uso una coda FIFO:

- All'inizio la coda contiene solo s.
- Al generico passo prelievo il nodo j, visito tutti i vertici adiacenti a j che non siano visitati e li metto nella coda. In questo modo i nodi visitati per primi sono i più vicini al nodo iniziale.

Pseudocodice:

```
breadth_first (vertex) {
	visit(vertex);
	enqueue(vertex);
	while( la coda non e' vuota) {
		x = dequeue();
		for( ogni vertice w adiacente ad x e non ancora visitato ) {
			visit(w);
			enqueue(w);
		}
	}
}
```

L'ordine di visita dipende dall'ordine in cui sono memorizzati i nodi nella lista d'adiacenza. Nell'esempio, se la sorgente è 3, non è molto importante se visito prima 1,2, o 4 perché tanto sono tutti distanti uno. Visito 3, lo metto in coda, poi le tolgo e passo al livello successivo, prendo 1,2 e 4 li visito e li metto nella coda; poi uno per uno li tolgo dalla coda, prendo 1 e visito tutti i nodi adiacenti a 1 non ancora visitati (c'è solo il 5 e lo metto nella coda). Poi passo a 2 e 4 e 5 (ma ho già visitato tutti i nodi).



Fonte: testgroup.it

Visitando un grafo, partendo dalla visita posso realizzare un albero equivalente.

***Teorema**: L'algoritmo di visita in ampiezza permette di visitare tutti i nodi del grafo solo se questo è connesso.*

***Teorema**: L'algoritmo di visita in ampiezza di un grafo rappresentato come liste delle adiacenze ha complessità O(max(|V|,|E|)). Nel caso in cui il grafo sia rappresentato come matrice delle adiacenze l'algoritmo ha complessità O(|V|2).*

Nota: applicando questo algoritmo, è in genere più conveniente utilizzare le implementazioni con lista di adiacenza piuttosto che le matrici.

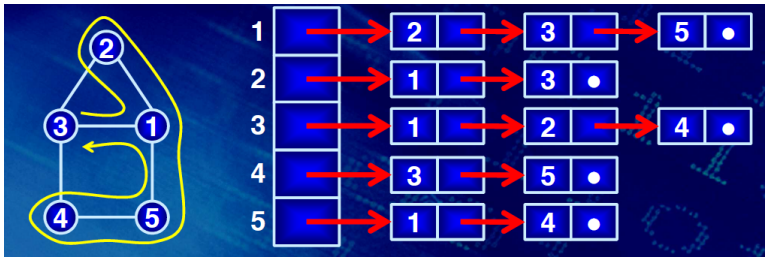
Visita in profondità

Qui non esploro livello per livello ma vado direttamente in profondità e poi risalgo. Al generico passo, dopo aver visitato il nodo k, prendo il nodo a lui adiacente (se c'è). Una volta che ho finito (sono arrivato a una "foglia") torno indietro di un livello e visito quelli non ancora esplorati.

```
Depth_first(vertex) {
      visit(vertex);
      for(ogni vertice w adiacente a w vertex non ancora visitato) {
                  depth_first(w);
      }
}
```

Nota: ogni volta che c'è la ricorsione, la posso sempre sostituire con un while. E vice versa.

Esempio:



Fonte: testgroup.it

La soluzione è 3-1-2-5-4.

È possibile implementarla in modo iterativo (come per la visita in ampiezza) sostituendo la FIFO con uno stack senza utilizzare la ricorsione (non tutti i linguaggi di programmazione ammettono la ricorsione).

Inizialmente nello stack c'è solo s, al generico passo prendo il nodo k e lo visito, poi cerco quelli a lui adiacenti non ancora visitati. Se ce ne sono proseguo la visita con questi, altrimenti se k non ha nodi adiacenti torno indietro al nodo che ha portato a k (che chiamo j) e guardo se ci sono nodi adiacenti a j non ancora visitati. Attraverso il meccanismo dello stack prendo sempre per primo l'ultimo che ho inserito.

***Teorema**: L'algoritmo di visita in profondità di un grafo rappresentato come liste delle adiacenze ha complessità O(|E|) nella versione ricorsiva e O(max(|V|,|E|)) in quella iterativa. Nel caso in cui il grafo sia rappresentato come matrice delle adiacenze l'algoritmo ha complessità O(|V|^2).*

Anche qui l'implementazione con le liste d'adiacenza sembra la migliore.

Nota: se un grafo è un albero e il nodo s è la radice dell'albero, una visita in profondità del grafo è equivalente a una visita pre-order dell'albero. La visita in profondità è particolarmente adatta per risolvere problemi di tipo labirinto.

Esempio: visita in profondità e in ampiezza partendo da A.
Se non viene fornita la lista d'adiacenza, se devo farlo a mano senza pesi e se non ho vincoli di altro genere, se ho più percorsi possibili devo scegliere quello che massimizza il numero di vertici toccati.
Può succedere che dei nodi non siano raggiungibili.
Visitando un grafo costruiamo implicitamente un albero (in profondità è sempre vero, in ampiezza non è detto) i cui archi sono un sottoinsieme degli archi del grafo chiamati **tree edges**.

Se il grafo non è orientato, gli archi che non sono dei tree edges connettono un nodo con un suo predecessore nell'albero di visita, e per questo prendono il nome di **back edges**.

Se il grafo è orientato invece, abbiamo quattro possibili situazioni:

- **Tree edges**: archi che appartengono all'albero di visita.
- **Back edges**: non appartengono all'albero di visita e connettono un nodo con un suo predecessore.
- **Forward edges**: non appartengono all'albero di visita e connettono un nodo con il suo successore.
- **Cross edges**: archi non appartenenti a un albero di visita e connettono due nodi che non sono un il predecessore dell'altro.

Il predecessore non è il nodo che viene prima nella sequenza di visita ma il padre nel senso della connessione.

Graphs: shortest paths & spanning trees

Cammini e grafi ciclici

***Def**. Un **cammino** in un grafo è una sequenza di vertici (v_0,v_1,v_2…) per cui esite un arco che collega ogni coppia di vertici successivi nella sequenza.*

Si definisce ***lunghezza del cammino** la somma dei pesi associati agli archi da cui esso è composto*. Se un grafo non è pesato, allora ogni arco ha peso 1. In questo caso la lunghezza minima è anche quella con il minor numero di vertici toccati.

Un cammino è detto ***semplice*** se non usa uno stesso arco due o più volte, vale a dire se non contiene cicli. Cammino elementare: non tocca lo stesso vertice due o più volte.
Un ciclo o circuito è un cammino semplice per il quale i vertici di partenza e di arrivo coincidono.

Grafo aciclico: grafo che non contiene cicli.

Grafi diretti aciclici: grafo orientato che non contiene cicli. Spesso sono chiamati DAG (Direct Acyclic Graph) -> sono più generali degli alberi ma meno dei grafi. Vengono utilizzati per rappresentare la struttura sintattica di espressioni (matematiche o anche per i compilatori). Non sono alberi perché ci sono connessioni non permesse per un albero.

Come faccio a dire se un grafo è ciclico?

Se il grafo è non orientato, è ciclico se:

- Visitandolo in profondità, si ha un vertice adiacente diverso dal padre.
- Visitandolo in ampiezza, si visita un vertice già visitato.

Se il grafo è orientato, se e solo se ci sono dei back edges.

Teorema: *un grafo non orientato nel quale il grado di ogni nodo è >= 2, possiede almeno un ciclo.*
***Corollario**: un grafo non orientato di n nodi nel quale il grado di (n/2)+1 nodi è >=2 possiede almeno un ciclo.*

Vertici connessi: due vertici di un grafo (orientato o no) sono connessi se **esiste un cammino fra i due** (non è detto che sia un arco, possono anche non essere adiacenti, il cammino è qualcosa di più generale).

Vertici raggiungibili: *fissato un vertice v_i, l'insieme dei vertici tali per cui esiste un cammino tra v_i e v_j costituisce l'insieme dei vertici raggiungibili da v_i.*

Teorema: in un grafo qualunque con |V| vertici, se due vertici sono connessi allora esiste tra loro un cammino di lunghezza <= |V|-1.

Grafo connesso

Un grafo non orientato si dice connesso se presi due qualsiasi vertici a esso appartenenti, allora esiste un cammino fra questi.

Per i grafi orientati:
Grafo fortemente connesso: se presi due qualunque vertici, esiste sia un percorso di andata sia di ritorno fra questi. In pratica un grafo orientato è fortemente connesso se da un qualunque vertice è possibile raggiungere ogni altro vertice.
Grafo debolmente connesso: se il grafo equivalente non diretto (non orientato) è un grafo connesso.

Alberi

L'albero è un caso particolare del grafo, l'albero è un grafo connesso e aciclico.

***Teorema**: sia T un grafo con |V| nodi. Allora sono fatti equivalenti:*

- *T è un albero.*
- *T non contiene cicli ed ha |V|-1 archi.*
- *T è connesso e ha |V|-1 archi.*

T non contiene cicli ma aggiungendo un arco fra due nodi qualsiasi si crea esattamente un ciclo.

Un grafo le cui componenti connesse sono alberi è detto foresta. È come se avessi tanti alberi non connessi messi insieme.

Cammini minimi

Def. Un cammino è minimo se qualsiasi altro cammino tra gli stessi vertici ha una lunghezza maggiore o uguale. Il termine lunghezza e' da intendersi come somma di pesi e non come numero di archi utilizzati.

Fissato un vertice s, l'insieme dei cammini minimi tra s e qualsiasi altro vertice v da esso raggiungibile costituisce un albero, detto ***albero dei cammini minimi*** a partire da s.

Teorema: Siano T un albero la cui radice coincide con il vertice s, dist(s,x) la lunghezza del cammino in T tra s e il vertice x, lenght(x,y) il peso associato all'arco tra i vertici x e y. Allora:

T è un albero dei cammini minimi ⇔ dist(s,v)+lenght(v,w) >= dist(s,w), ∀(v,w) adiacenti.

Nota: in genere lenght() viene utilizzato per due vertici adiacenti, è il peso dell'arco, mentre dist è lunghezza del cammino (somma dei pesi) fra due nodi.

Vedremo due algoritmi per determinare i cammini minimi a partire da un vertice s:

Algoritmo basato su breadth-first

In questo algoritmo:

• dist[1:|V|] contiene le distanze di ogni nodo dal nodo di partenza s.

• parent[1:|V|] contiene l'indice del nodo padre.

• la coda utilizzata può essere indifferentemente FIFO o LIFO.

• per semplificare il test di presenza nella coda si può utilizzare un vettore isinqueue[1:|V|].

```
dist[s] = 0;
for(w!=s) dist[w] = MAXINT;
enqueue(s);
while ( la coda non e' vuota) {
      head = dequeue();
      for (ogni successore w di head)
      if ( dist[head] + length(head,w)< dist[w]) {
            dist[w] = dist[head] + length(head,w);
            parent[w] = head;
            if( w non e' nella coda)      enqueue(w);
      }
}
```

Head è la sorgente dalla quale abbiamo preso w (il predecessore, il padre di w, quello che ci ha permesso di raggiungere w). Si basa sulla disuguaglianza del teorema, con questo trovo la lunghezza minima dalla sorgente a qualsiasi nodo del grafo. Vado avanti trovando cammini alternativi passando da altri nodi, se queste lunghezze sono più piccole di MAX_INT (valore iniziale) o di quella attuale allora ho trovato un cammino più breve.

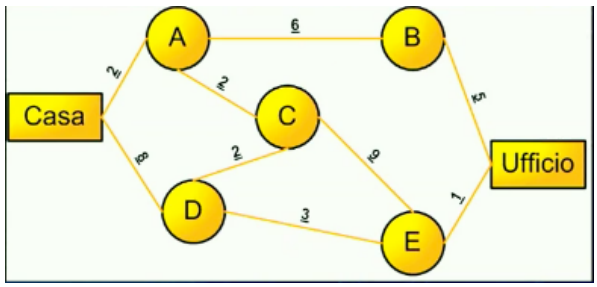
Posso avere cammini minimi equivalenti, quale scelgo? Se ho altri vincoli (ad esempio il minore numero di archi attraversati a parità di somma di pesi) uso quelli altrimenti lo scelgo in modo random (dal punto di vista della definizione sono del tutto equivalenti).

Algoritmo greedy di Dijkstra

```
S = empty set;
for( tutti i vertici w )      D[i] = C[1,i];
for( i=1, i++, i<n-1 ) {
      scegli un vertice w in V-S tale che D[w] sia minimo;
      inserisci w in S;
      for( tutti i vertici v in V-S )    D[v]= min(D[v],D[W]+C[w,v])
}
```

S è il set dei vertici già visitati, W è il set dei vertici totali. Parto dalla sorgente che è adiacente a 3 nodi, tra questi 3 non ancora analizzati scelgo quello che ha distanza minima. Se non uso questo accorgimento rischio di non ottenere il cammino minimo.

Ai nodi devo sempre dare un'etichetta (quanto è distante il nodo dalla sorgente) e devo sempre chi è il padre. Parto dalla sorgente (CASA) e illumino i suoi adiacenti (li metto nella FIFO), sono A e D, e fra questi calcolo il percorso dai nodi adiacenti dalla sorgente (in questo caso 2 e 8) e scelgo come prossimo nodo quello che ha distanza minima e dico che S è il predecessore di A. Poi itero e vado a finire in C. A questo punto gli adiacenti di C sono E e D, ma nel caso di D avevo come distanza dalla sorgente 8, però esplorando ci siamo accorti che c'è un altro cammino minimo tra casa e D e quindi aggiorno la dist fra casa e D (sfrutto la disuguaglianza, (8<4+2 -> no…). Se in un generico nodo ritrovo come adiacente CASA, la sua dist non la tocco perché quello è la mia sorgente e rimane fisso. Anche per E accade lo stesso che per D perché di nuovo imbatto in quella disuguaglianza. Quando parlo di predecessore non parlo del nodo che ha illuminato il nodo che illumino, ma quello che istante per istante attraverso lui ottengo il cammino minimo fino a quel nodo con le informazioni che ho a disposizione.



Una volta che ho raggiunto la destinazione, non è detto che il cammino minimo sia quello che ho appena trovato, se ho dei nodi non ancora esplorati li esporo. Una volta trovato il cammino minimo ripercorro i predecessori e così ottengo il percorso minimo.

Algoritmo greedy perché non cerco la soluzione ottima a livello globale (perché non c'è l'ho) ma perché passo per passo cerco la soluzione ottimale. Questi sono chiamati algoritmi distribuiti perché i singoli nodi vedono solo il loro contenuto e poche informazioni che gli arrivano, con questo tipo di distribuzioni si applica una politica greedy -> a ogni step scelgo la soluzione ideale. Non avendo però una visione globale, potrei rischiare di trovare una soluzione che non è ottimale a livello globale. Sono leggeri perché lavorano su poche informazioni, non ho ogni volta non ho tutte le informazioni ma quelle poche che mi servono.

Analisi della complessità: se uso una coda prioritaria allora la complessità è un O(|E| + |V| log |V|). In ogni caso non è né quadratica né esponenziale -> va bene anche quando il set di nodi è molto elevato.

Determinazione del cammino minimo tra tutte le coppie: presi due nodi a caso nel grafo (1 è la sorgente e l'altro è la destinazione) voglio trovare il cammino minimo fra questi due. Posso o fissare s e facendo variare i vertici di destinazione (applicando uno dei due algoritmi precedenti) e iterare

questo procedimento per tutte le coppie, oppure utilizzare un algoritmo basato sulle matrice di adiacenza (algoritmo di Floyd).

Albero ricoprente

*Def. dato un grafo G non orientato viene definito **albero ricoprente (spanning tree)** quel sottografo che è un albero e contiene tutti i vertici di G.*

Nel caso dei cammini minimi potrei avere dei nodi che non considero.

***Teorema:** ogni grafo non orientato e connesso contiene almeno un albero ricoprente.*

Dato un sottografo G' di un grafo G non orientato e pesato, si definisce peso di G' la somma dei pesi degli archi appartenenti a G'. Si definisce **minimum spanning tree** di un grafo lo spanning tree avente peso minimo.
Inoltre posso avere più di un minimum spanning tree e sono tutti equivalenti fra di loro.
In genere se parto da un grafo e tolgo un po' di archi ottengo un albero.

La determinazione del minimum spanning tree è importantissima in tutte in quelle applicazioni dove devo minimizzare il numero di interconnessioni tra gli elementi. Se ho un grafo molto complesso e molto connesso, potrei per fare delle operazioni dover cercare un albero ricoprente minimo.

Algoritmo di Dijkstra-Prim

Qui non c'è più la sorgente e la destinazione, prendo un elemento arbitrario da usare come radice (dovrà avere almeno un elemento adiacente).
Viene utilizzata una suddivisione dei vertici del grafo in tre sottoinsiemi disgiunti:

> **Tree vertices**: vertici già appartenneti all'albero.
> **Fringe vertices**: vertici non appartenenti all'albero ma adiacenti ad almeno uno dei tree vertices.
> **Unseen vertices**: tutti gli altri.

```
seleziona un vertice arbitrario come radice;

while (ci sono fringe vertices){
      seleziona uno degli archi di peso minimo tra quelli che collegano un
      tree vertexe un fringe vertex;

      aggiungi l'arco selezionato ed il fringe vertex relativo all'albero;

      aggiorna di conseguenza i tre insiemi di vertici;
}
```

Nel caso in cui avessi dei cicli, devo tagliare i cicli tenendo quello a cammino minimo. Aggiungo all'albero sia l'arco che pesa di meno sia il nodo relativo a quell'arco, e poi prendo anche i nodi adiacenti del nodo appena aggiunto all'arco (se questi nodo sono già presenti nei fringe vertices, allora scelgo come arco quello con peso minore).

Quando le dimensioni sono molto ampie gli algortimi greedy permettono di ottenere in tempi brevi ciò che volevo. Procedo scegliendo l'ottimo locale e non globale. In questi due casi con gli algoritmi

greedy ottengo magicamente anche la soluzione ottimale a livello globale (non è sempre detto ma per i due Dijkstra è vero).

Può essere conveniente memorizzare l'insieme degli archi che collegano un tree vertex a un fringe vertex in una coda prioritaria.

Il costo computazionale è O(|V|^2). Ho un |V| per tutti i nodi del grafo (il while esterno) e poi devo scegliere quello minimo che nel worst case è |V|-1. Le istruzioni che pesano 1 sono irrilevanti, ciò che invece pesa sono quelle istruzioni che scandiscono tutta quanta la lista.

Graphs: Euler & Hamilton Graphs

Grafi euleriani

Def. ***Cammino di Eulero****: cammino che percorre tutti gli archi di un grafo esattamente una e una sola volta* (se il grafo è complesso può essere difficile da trovare) (qui sorgente e destinazione sono in genere diversi).

Ciclo di Eulero: ciclo che percorre tutti gli archi una sola volta (sorgente=destinazione).

Esempio: ponte di Konigsberg, è possibile attraversare i 7 ponti una sola volta? Oppure disegnare una figura senza staccare la matita dal foglio e senza ripassare due volte per la stessa riga?

Grafo euleriano: è un grafo connesso che contiene un ciclo di Eulero. Ovvero con alcuni vertici, riesco a percorrere tutti gli archi una sola volta e tornare al vertice di partenza. ***Grafo semi-euleriano***: grafo connesso che contiene un cammino di Eulero.

È possibile avere delle condizioni necessarie e sufficienti affinché' un grafo sia euleriano?

Teorema*: un grafo non orientato è euleriano se e solo se è connesso e i suoi vertici sono tutti di ordine pari.*

Corollario: un grafo non orientato è semi-Euleriano ⇔ è connesso e possiede o nessuno o esattamente due vertici di ordine dispari.

Corollario*: Un grafo orientato possiede un cammino di Eulero se e solo se è connesso e, per ogni vertice (con la possibile eccezione di due), il grado in ingresso è uguale al grado in uscita. Per gli eventuali due vertici anomali, in uno il grado in ingresso deve essere di 1 superiore al grado in uscita, mentre nell'altro il grado in ingresso deve essere di 1 inferiore al grado in uscita.*

Corollario*: un grafo orientato possiede un ciclo di Eulero se e solo se è connesso e per ogni vertice il grado in ingresso è uguale al grado di uscita.*

Proviamo a costruire un cammino di Eulero attraverso l'algoritmo di Fleurry in un grafo. Con i teoremi so se un grafo è euleriano o no, però non so qual è il ciclo. Con questo algoritmo ricorsivo lo trovo:

- Trovare un ciclo semplice in un grafo connesso, avente m+1 archi.

– Cancellare gli archi del ciclo e trovare un ciclo di Eulero in ciascuna delle componenti risultanti.

– Unire i cicli di Eulero in un ciclo semplice unico, ottenendo così il ciclo di Eulero cercato.

Dobbiamo trovare i cicli, e poi unirli a patto che scegliamo un solo punto d'intersezione comune. Devo mettere un numero sugli archi perché così ho l'ordine di percorrenza. Dopo che collego un ciclo rinumero gli archi in modo da legare il ciclo e mantenere qualcosa di coerente.

Per trovare facilmente un cammino, posso dire: ok non c'è un ciclo e aggiungo un arco fantasma in modo da avere il ciclo, trovo la sua percorrenza e poi cancello l'arco fantasma. Ciò che ottengo è un cammino.

Grafi hamiltoniani

Def. ***Cammino di Hamilton****: posso passare sugli archi quante volte voglio ma devo toccare tutti i vertici esattamente una sola volta.*

Se un grafo ha |V| vertici allora il cammino di Hamilton contiene esattamente |V|-1 archi.

Ciclo di Hamilton: ciclo che percorre tutti i vertici di un grafo esattamente una sola volta (in pratica parto dalla sorgente e torno alla sorgente, è vero che tocco il nodo sorgente due volte ma questo è implicito nel concetto di ciclo).

Ad oggi non esiste una dimostrazione che permetta di stabilire se un grafo è Hamiltoniano o meno. È stato dimostrato che l'individuazione dell'esistenza di un cammino di Hamilton è un problema NP hard (non c'è soluzione in un tempo finito, il problema letteralmente esplode con n->oo).

Teorema di Dirac (condizione sufficiente ma non necessaria) sia G un grafo semplice con |V| nodi. Se |V|>=3 e $\rho(v)>=|V|/2$ per ogni nodo v allora G è Hamiltoniano.

Poiché un cammino di Hamilton passa attraverso un vertice una e una sola volta, solo 2 degli archi che incidono su un vertice possono essere inclusi in un cammino. Condizione necessaria ma non sufficiente per l'esistenza di un cammino di Hamilton è che:

$$\Sigma_1 - \Sigma_2 > |E|$$

dove:

- Σ_1 è la somma dei gradi complessivi dei vertici del grafo
- Σ_2 è la somma degli archi che non possono essere inclusi in un cammino.