HC – HasCycle

Progetto 2 - Algoritmi di verifica della presenza di un ciclo in un grafo non orientato connesso.

Si presenta l'implementazione dell'algoritmo hasCycle nelle sue versioni DFS e UF, insieme a un algoritmo che genera grafi non orientati connessi e un decoratore con ruolo di profiler.

Mihai Jianu, Daniele La Prova, Lorenzo Mei
[IA1819]

Indice

- Copertina.....1
- Indice.....2
- Strategia di Implementazione.....3
  - Implementazione di GG 3
  - Implementazione di HCDFS.....4
  - Implementazione di HCUF.....6
  - Implementazioni di P.....7
- Raccolta ed elaborazione dati.....8
  - Grafici e Tabelle.....8

Strategia di Implementazione

Per adempiere alle richieste presenti nella Traccia, è stata adottata la strategia di implementazione descritta di seguito:

- Implementazione dell'algoritmo `gGenerator` (**GG**), che genera grafi non orientati connessi (a)ciclici rappresentati come matrice di adiacienza;
- Implementazione dell'algoritmo `hasCycle` (**HC**), che verifica la presenza di un ciclo nel tipo di grafi sopracitati sfruttando la visita DFS (**HC_DFS**) o la struttura dati Union-Find (**HC_UF**);
- Implementazione del decoratore `@profiler` (**P**), che regsitra dati prestazionali delle funzioni su cui è applicato e li annota in un file di testo.

Implementazione di GG

Riguardo la rappresentazione dei grafi è stato deciso di utilizzare come struttura dati le matrici di adiacenza, in quanto il tempo di esecuzione richiesto per l'operazione utilizzata più frequentemente da HC, ovvero la verifica dell'adiacenza tra due nodi, richiede tempo $O(1)$. Questo vantaggio è ottenuto al prezzo di un impiego di memoria dell'ordine di $O(n2)$, che limita la dimensione dei grafi intorno ai 10000 nodi.

ATTENZIONE: un tentativo di generazione di un grafo con un numero di nodi $n > 10000$ può comportare seri rallentamendi della macchina, fino al freeze totale. Per tale ragione, il valore di n è limitato dalla costante $MAXNODE = 10000$.

L'algoritmo dietro al funzionamento di GG consiste in pochi, semplici passi:

- $graph \leftarrow GraphAdjacencyMatrix()$
- $tailList \leftarrow getNodes(graph)$
- for $n - 1$times do:
  - $tail \leftarrow tailList$.pop(random index)
  - $head \leftarrow tailList$[random index]
  - inserisci l'arco (tail, head) e l'arco (head, tail) in $graph$
- aggiungi $cycle$ archi tra nodi non adiacenti tra loro
- return $graph$

In altre parole, GG prende in input il numero di nodi n, il range di rappresentazione dei valori dei nodi $rangeG$ e il numero di cicli richiesto $cycle$. Per $n - 1$ volte, esegue un pop di un nodo tail casuale da una $tailList$ composta inizialmente da n nodi. Inoltre, sceglie un altro nodo $head$ casuale dalla $tailList$ (senza eseguire il pop) e inserisce nel grafo gli archi $(tail, head)$ e $(head, tail)$. In questo modo si ottiene un grafo non orientato connesso con n nodi e $2n - 2$ archi. Inoltre, aggiunge $cycle$ archi al grafo tra nodi non adiacienti tra loro, inserendo dunque $cycle$ cicli.

Il tempo di esecuzione previsto è $O(n - 1)$ se il grafo richiesto è aciclico.

Se è richiesta l'aggiunta di cicli, il tempo di esecuzione sale a $O(n-1+cycle)$, con $cycle = C_{n,2} - (n-1) = \frac{n!}{2(n-2)!} - n + 1 = \frac{1}{2}(n^2 - 3n + 2) = O(n^2)$ nel caso peggiore.

L'implementazione di GG è scritta nel modulo `GG_module.py`.

Implementazione di HCDFS

Il file `HCDFS_module.py` contiene l'implementazione dell'algoritmo `HCDFS` e le relative funzioni che consentono il suo funzionamento.

Partendo dalla funzione `haCycleDFS()`, essa assume in input il parametro `G`, ovvero il grafo, ed altri dati opzionali quali:

- `debug`, booleano che permette di visualizzare informazioni in più per il debugging;
- `showProfile`, per la visualizzazione a schermo delle informazioni che vengono scritte dal Decorator all'interno del file (Vedi sezione D_module).

Al suo interno viene creata un'istanza di `CustomGAM` (**CGAM**), ovvero una sottoclasse della classe `GraphAdjacencyMatrix`. Si è scelto di implementare questa sottoclasse per non alterare la struttura del metodo `dfs()` presente in `Graph.py` per la rilevazione di cicli nei grafi.

In seguito viene chiamato il metodo `dfsDetectedCycle()` presente in CGAM, che prende in input i parametri `rootId` (nodo di partenza della visita DFS) e `debug`. Esso controllerà attraverso la visita DFS se è presente o meno un ciclo in G.

La visita DFS viene implementata attraverso uno stack e vengono aggiunti al suo interno tutti i nodi adiacenti al vertice preso in considerazione che non sono ancora stati esplorati. Questo vuol dire che se un nodo (A) è presente all'interno dello stack almeno 2 volte, allora esso è stato "visto" da due nodi diversi ma a lui adiacenti (B, C adiacenti ad A). Perché questo dovrebbe implicare la presenza di un ciclo? Perché B e C sono connessi tra loro o attraverso un arco (Figura 1) oppure attraverso un altro nodo (Figura 2). Infatti se ci siamo fermati sul nodo B, abbiamo "visto" tutti i suoi nodi adiacenti, abbiamo messo A nello stack ed in seguito siamo passati al nodo C, allora esso deve necessariamente essere connesso a B, direttamente o attraverso un altro vertice, in quanto il nodo successivo da esaminare è scelto attraverso un pop dalla pila stessa.

Quindi viene effettuato un controllo sullo stack ed attraverso un contatore si esamina quante volte è presente un dato nodo al suo interno.

Questa scelta nella risoluzione del problema evita molti inconvenienti, uno dei tanti ad esempio è la confusione di un arco per un ciclo (essendo il grafo non orientato), in particolare dell'arco incidente al nodo in cui ci troviamo ed il vertice esaminato precedentemente.

Il tempo di esecuzione è analogo a quello di una vista DFS su grafi implementati come matrici di adiacienza, ovvero $O(n^2)$.

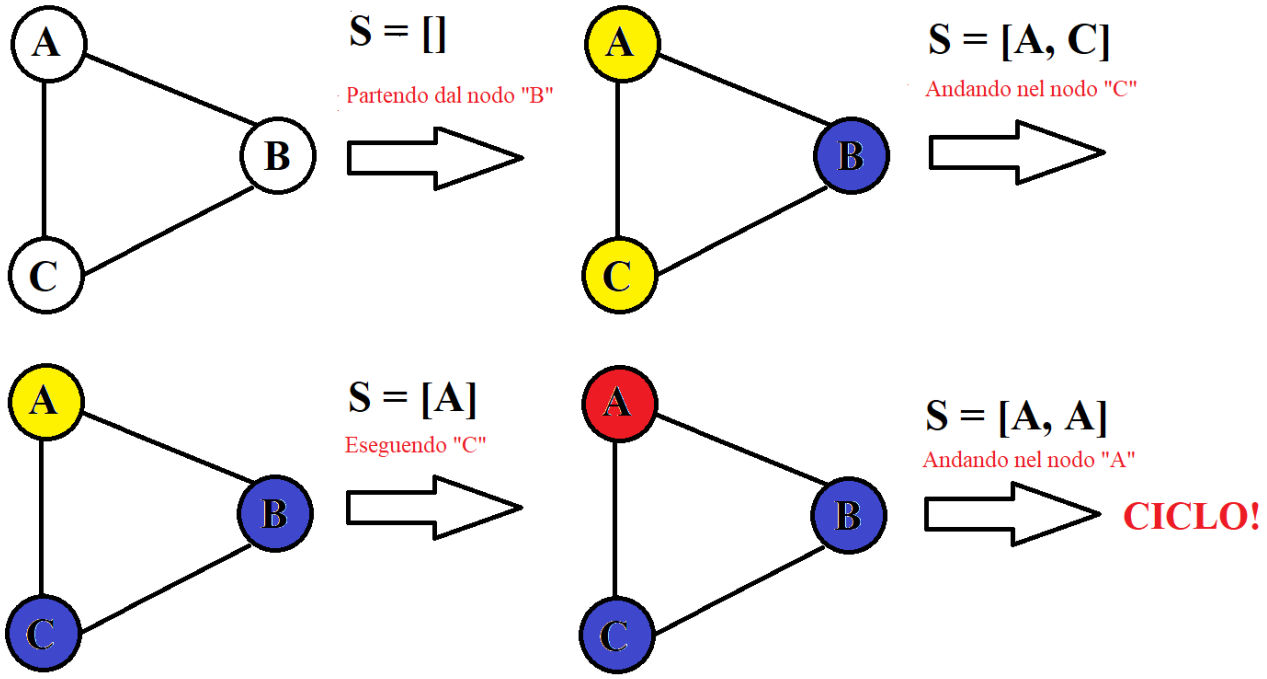

Figura 1. Caso di ciclo in cui B e C sono connessi tramite un arco

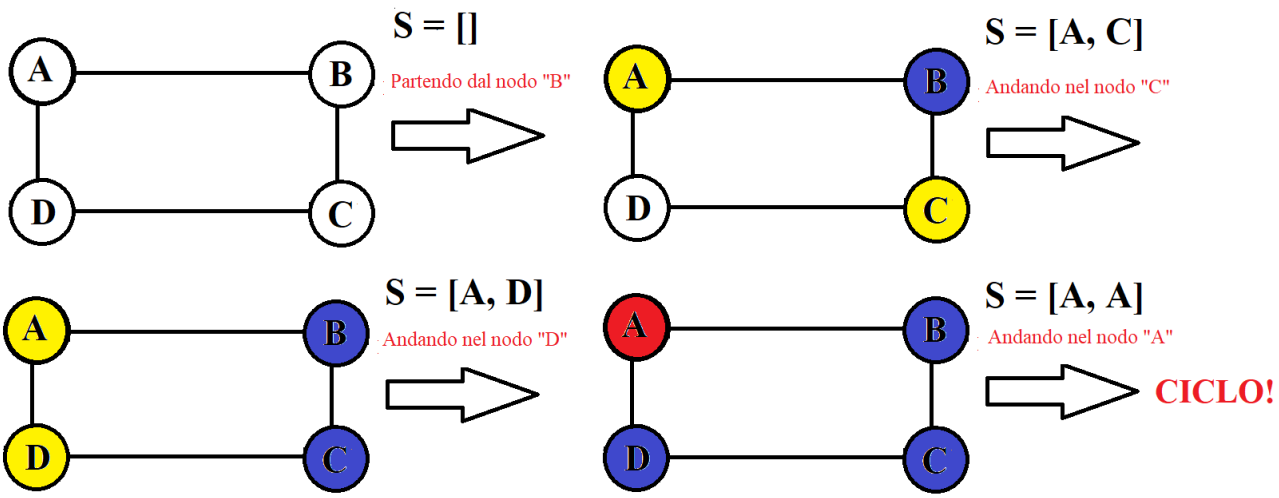

Figura 2. Caso di ciclo in cui i nodi D e B sono connessi tramite un altro nodo C

L'algoritmo HCUF, oltre ai parametri opzionali `debug` e `showProfile`, prende come input un grafo `G` e verifica se al suo interno sia presente almeno un ciclo sfruttando la struttura dati Union-Find. Tra le varianti disponibili è stato scelto di implementare la versione QuickFindBalanced, in quanto velocizza al massimo l'operazione di find riducendo al minimo l'impatto prestazionale sull'operazione di union. Inoltre, analizzando lo pseudocodice fornito nella Traccia si può notare come a ogni passo vengano effettuate due operazioni di find e una di union. Dunque, essendo la find l'operazione più frequentemente impiegata, si è rivelata come la più meritevole di ottimizzazione, a scapito della union.

L'algoritmo scorre tutti gli archi del grafo usando un iterabile generato dalla funzione `edgeGenerator()`. Ad ogni passo, controlla se i valori della coda e della testa dell'arco corrente siano presenti all'interno di un set della Union-Find, e nel caso crea dei set che contengono come unico elemento tali valori (uno per set). Inoltre, richiama `edgeGenerator()` per generare un iterabile che scorre gli archi finora visitati, in modo da poter controllare se l'arco corrente sia già stato esaminato nel senso opposto. Se non si trova in tale caso, e la testa e la coda dell'arco si trovano nello stesso set, allora l'algoritmo ha trovato un ciclo e termina l'esecuzione, altrimenti tenta di unirne i set e continua con l'esecuzione.

Data la natura dell'implementazione della struttura dati Union-Find è stato necessario definire una nuova classe `CustomQFB(QuickFindBalanced)` che implementasse il metodo findNode. Tutti i metodi forniti da QuickFindBalanced richiedono in input un nodo UFBNode, dunque è stata ritenuta necessaria l'implementazione di un metodo `findNode()` che, preso in input un elemento, restituisse il nodo nella Union-Find che lo contenesse. Sfruttando il fatto che gli elementi corrispondono agli ID dei vertici di un grafo e che ogni ID è unico si evita di incorrere in ambiguità, poiché tutti i nodi contengono elementi diversi tra loro.

Il caso peggiore si ottiene nel caso in cui il grafo sia connesso aciclico, dunque disponga di $n-1$ archi. Il tempo di esecuzione in tale caso è $O((n-1)\log(n))$.

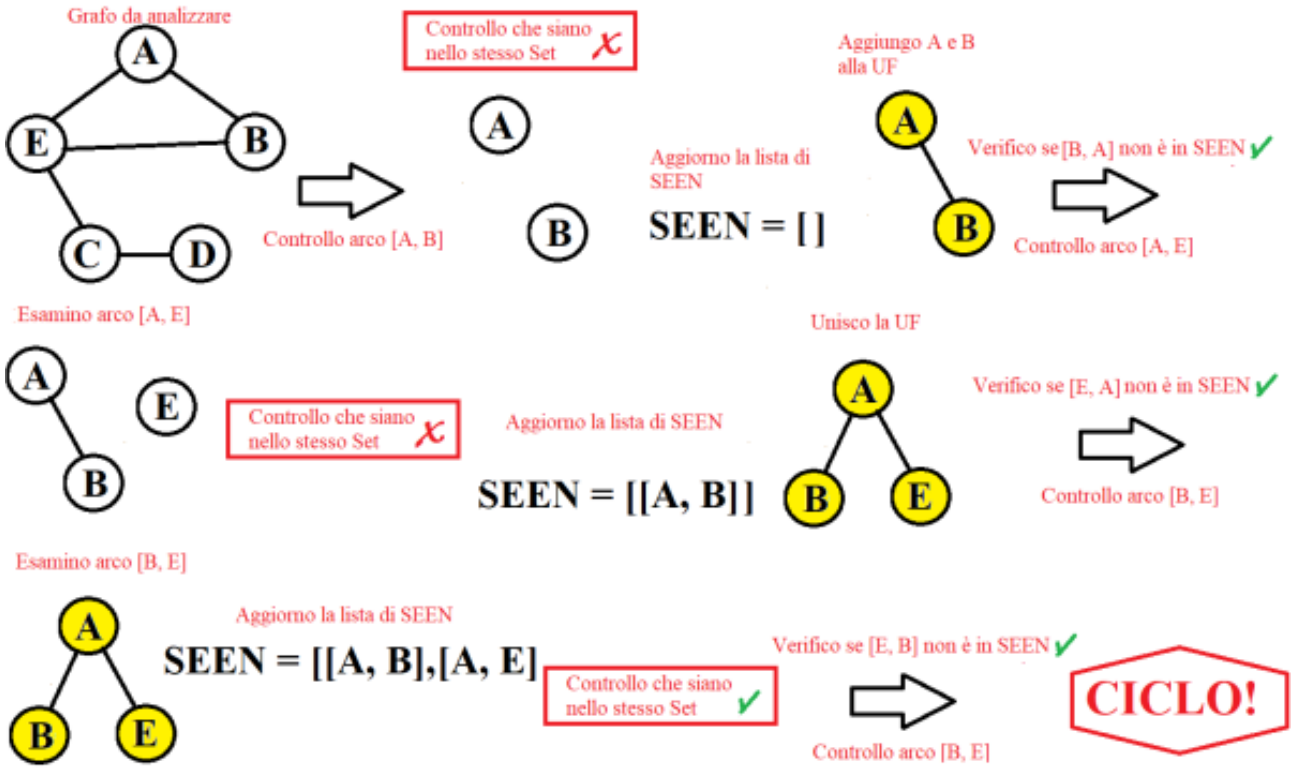

Figura 3. Esempio di applicazione dell'algoritmo HCUF su grafo ciclico di 5 nodi.

Implementazione di P

I dati prestazionali di HCUF e HCDFS sono stati raccolti applicando il decorator P a tali funzioni. Il decoratore si preoccupa di annotare i dati prestazionali delle funzioni, specificando:

- La data dell'esperimento;
- Il nome della funzione eseguita;
- Il numero di nodi e archi del grafo di input;
- Il tempo di esecuzione;
- Il valore di ritorno.

Se il parametro opzionale `showProfile`, presente nelle definizioni di HCUF e HCDFS, è impostato come `True`, i dati prestazionali saranno anche stampati a schermo.

Di default, i dati prestazionali sono scritti nel file `log.txt`. Tuttavia, è possibile specificare un percorso diverso utilizzando il parametro opzionale `pathLog`, sempre nelle definizioni di HCUF e HCDFS.

A causa della natura "mistificatrice" dei decoratori in Python, per permettere di annotare nel file di log il nome corretto della funzione da testare è stato necessario applicare a P il decoratore `include_stripped()`.

Raccolta ed Elaborazione Dati

In questa sezione ci occuperemo dell'analisi dei dati, la raccolta di essi tramite il decoratore, l'elaborazione attraverso grafici e tabelle ed infine il loro studio.

Il supporto fisico su cui sono stati effettuati i test aveva le specifiche descritte in Figura 4.

Le misurazioni ottenute sono frutto di una media sviluppata in base al risultato di almeno 3 test.

Nel caso di grafi ciclici il numero di cicli è pari a 1.

HP 15-ay065nl
Processore Intel® Core™ i5-6200U(2.3/2,8GHz, 3MB L2)
HDD 1000 GB - RAM 16GB - Display 15,6" WLED Full HD
Wi-Fi 802.11a/b/g/n/ac - Bluetooth 4.2 - Windows 10
Scheda grafica AMD Radeon R5 M430(2GB Dedicata)

Figura 4. Specifiche tecniche del supporto fisico utilizzato nella fase di testing

Grafici e Tabelle

I primi due grafici (Figura 5 e Figura 6) mostrano i tempi d'esecuzione dell'algoritmo HCDFS al variare della dimensione del grafo dato in input. Si nota banalmente come la presenza di un grafo con cicli comporti un tempo d'esecuzione minore, questo perché la funzione appena ne trova uno termina il suo lavoro, altrimenti dovrà scansionare tutti i vertici. Con un grafo di 500 nodi ciclico il tempo di HCDFS è di poco più di 0.03 secondi mentre quello aciclico impiega 0.06 secondi. Con un grafo, invece, di 10000 nodi il tempo d'esecuzione arriva a circa 10 secondi per il grafo ciclico e più di 25 secondi per quello aciclico.

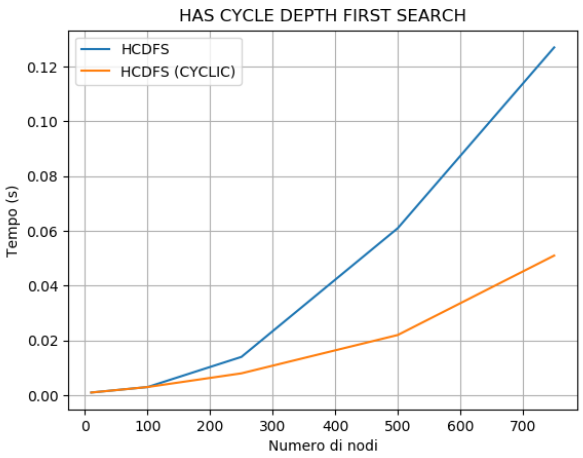

Figura 5. Tempo d'esecuzione di HCDFS per grafi ciclici e aciclici

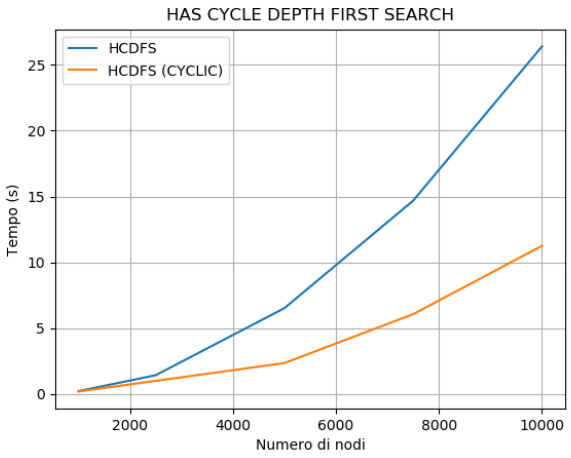

Figura 6. Tempo d'esecuzione di HCDFS per grafi ciclici e aciclici

Algorithm	Number of Nodes				
HCDFS	10	100	250	500	750
Cyclic	0,001	0,003	0,008	0,022	0,051
Acyclic	0,001	0,003	0,014	0,061	0,127

Figura 7. Tabella dei tempi d'esecuzione di HCDFS

Algorithm	Number of Nodes				
HCDFS	1000	2500	5000	7500	10000
CYCLIC	0,198	1,008	2,356	6,072	11,256
ACYCLIC	0,225	1,436	6,526	14,689	26,391

Figura 8. Tabella dei tempi d'esecuzione di HCDFS

I grafici seguenti (Figura 9 e Figura 10), invece, mostrano i tempi d'esecuzione per l'algoritmo HCUF. Si nota da subito la differenza con l'algoritmo HCDFS, infatti con grafi in cui il numero di nodi è pari a 700 il tempo d'esecuzione di HCUF è di 63,325 secondi a differenza dei 0,51 secondi impiegati da HCDFS. Con grafi contenenti una quantità elevata di nodi il tempo impiegato nella rilevazione del ciclo è molto elevato, arrivando a 16'427,111 secondi. A causa di ciò le misurazioni per grafi con 5000 nodi aciclici, con 7500 e 10000 nodi non sono state effettuate, è stata realizzata perciò una stima approssimativa del loro tempo d'esecuzione grazie alla formula che descrive il caso peggiore di HCUF.

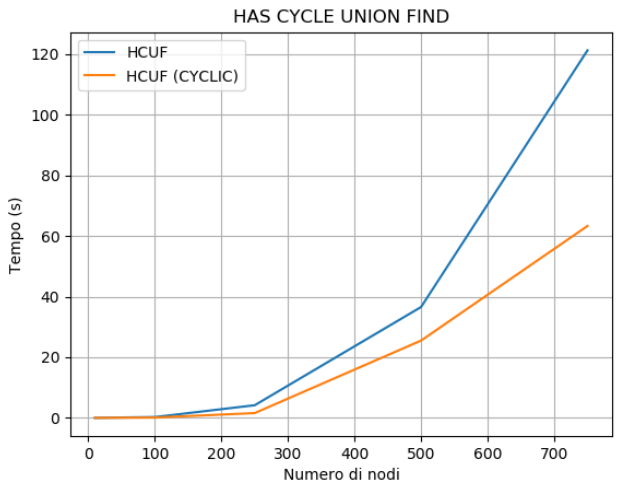

Figura 9. Tempo d'esecuzione di HCUF per grafi ciclici e aciclici

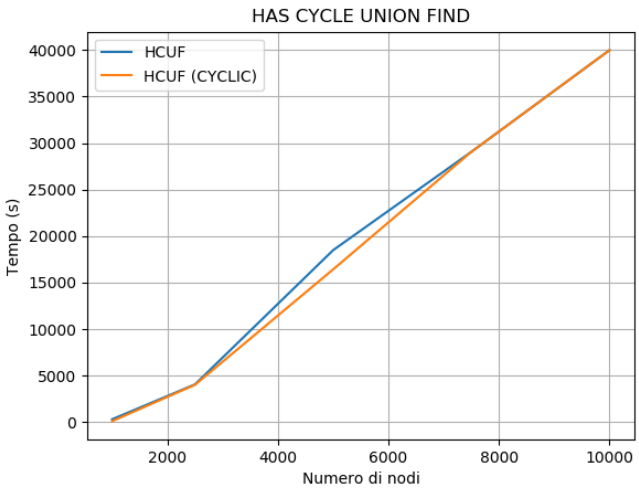

Figura 10. Tempo d'esecuzione di HCUF per grafi ciclici e aciclici

Algorithms	Number of Nodes				
HCUF	10	100	250	500	750
Cyclic	0,001	0,098	1,579	25,509	63,325
Acyclic	0,001	0,269	4,172	36,586	121,271

Figura 9. Tabella dei tempi d'esecuzione di HCUF

Algorithms	Number of Nodes				
HCUF	1000	2500	5000	7500	10000
Cyclic	100,954	4'010,093	16'427,111	*29'059,084	*39'996,000
Acyclic	299,862	4.054.008	*18'491,151	*29'059,084	*39'996,000

*Figura 10. Tabella dei tempi d'esecuzione di HCUF. Le misurazioni ottenute con una stima teorica sono indicate con *.*