Corso Magistrale in Ingegneria Informatica - Advanced Computing

Relazione dell'Elaborato

Software Engineering for Embedded Systems

Autrice: Agata Parietti

Data Esame: 14/02/2024

1 Introduzione

La relazione presente affronta il tema del testing del software, un aspetto cruciale nello sviluppo di applicazioni affidabili e di qualità. Il progetto in esame si concentra sull'analisi e il testing di codice scritto in linguaggio C, con particolare attenzione alla generazione di un Control Flow Graph e alla creazione di una Test Suite basata sul criterio "All Paths".

Il Control Flow Graph (CFG) è una rappresentazione grafica del flusso di controllo all'interno di un programma. In sostanza, mostra come il controllo del programma passa da un'istruzione all'altra durante l'esecuzione. Il CFG include nodi che rappresentano le istruzioni del programma e archi che rappresentano i flussi di esecuzione tra di esse, includendo rami condizionali, cicli e sequenze di istruzioni. La generazione di un CFG è un'attività fondamentale nell'analisi statica del software, in quanto fornisce una visualizzazione chiara della struttura del programma e delle possibili strade che può percorrere durante l'esecuzione. Questo è utile per comprendere il comportamento del software e identificare potenziali problemi o aree critiche che richiedono attenzione durante il processo di testing.

Il testing di tipo "All Paths" è una strategia di testing che mira a testare tutte le possibili strade di esecuzione attraverso il CFG. Questo significa che ogni percorso possibile, compresi i rami condizionali e i cicli, viene testato per verificare il comportamento del software in diverse condizioni di input. Questo approccio è particolarmente utile per identificare difetti e comportamenti imprevisti nel software, in quanto fornisce una copertura completa di tutte le possibili situazioni che il programma potrebbe incontrare durante l'esecuzione.

Nel contesto del progetto, oltre alla generazione di un CFG e alla creazione di una Test Suite basata sul criterio "All Paths", sono state sviluppate anche delle Test Suite per gli algoritmi di ordinamento scritti in linguaggio C. Questo implica la progettazione e l'esecuzione di una serie di test che coprono tutti i casi di input e le possibili combinazioni di dati, al fine di garantire che gli algoritmi di ordinamento funzionino correttamente in tutte le situazioni previste.

2 Stato dell'arte

Lo stato dell'arte evidenziato dall'articolo "Test Generation and Test Prioritization for Simulink Models with Dynamic Behavior" affronta una sfida significativa nel campo della verifica e del testing dei modelli Simulink con comportamento dinamico. Gli autori identificano tre sfide principali: incompatibilità, oracolo e scalabilità.

Il loro approccio di test generation si basa su una ricerca meta-euristica per massimizzare la diversità nei segnali di output generati dai modelli Simulink, un approccio particolarmente adatto per i sistemi cibernetico-fisici (CPS) che dipendono fortemente da modelli con comportamenti dinamici. Inoltre, l'articolo introduce un algoritmo di test prioritization che supera la prioritizzazione casuale dei test e un algoritmo di stato, insieme a uno strumento di testing chiamato SimCoTest.

Lo scopo iniziale del progetto era quello di adottare un approccio simile a quello descritto nell'articolo, con l'obiettivo di ricostruire un sistema di test per i modelli Simulink. Tuttavia, a causa di vincoli contrattuali e di accordi di non divulgazione, i sistemi testati dagli autori non potevano essere resi pubblici. Di conseguenza, abbiamo deciso di modoficare il progetto creando un programma capace di creare un CFG a partire da un sistema scritto in codice C e generare una Test Suite. Per testare il funzionamento dell'algoritmo proposto, abbiamo utilizzato semplici algoritmi di ordinamento come casi di test, al fine di valutare l'efficacia e la robustezza del nostro approccio nel contesto del testing software in linguaggio C. Questa decisione ci ha permesso di mantenere l'obiettivo principale del progetto, nonostante le limitazioni riscontrate nel reperire i modelli Simulink utilizzati nell'articolo di riferimento.

3 Implementazione

Il programma è stato sviluppato in linguaggio Java e si compone di cinque classi principali che svolgono ruoli specifici all'interno del processo di generazione e esecuzione dei test.

1. **Node**:
   La classe Node rappresenta un nodo all'interno del Control Flow Graph (CFG).

2. **Graph**:
   La classe Graph è responsabile della costruzione e della gestione del Control Flow Graph. Utilizzando oggetti Node, il Grafo viene creato in base alla struttura del codice sorgente.

3. **Parser**:
   La classe Parser svolge un ruolo cruciale nel processo di trasformazione del codice sorgente in una rappresentazione utilizzabile per la generazione dei test. Il suo compito è interpretare il codice sorgente, applicare regole di formattazione e trasformarlo in una struttura dati comprensibile dal programma. Questo aiuta a garantire coerenza e uniformità nel modo in cui il codice viene interpretato e analizzato.

4. **TestSuite**:
   La classe TestSuite è responsabile della generazione dei test a partire dal Control Flow Graph. Utilizzando informazioni fornite dal Grafo, crea una serie di casi di test che coprono tutte le possibili strade di esecuzione all'interno del programma.

Il progetto è strutturato come illustrato dal diagramma UML sotto riportato.

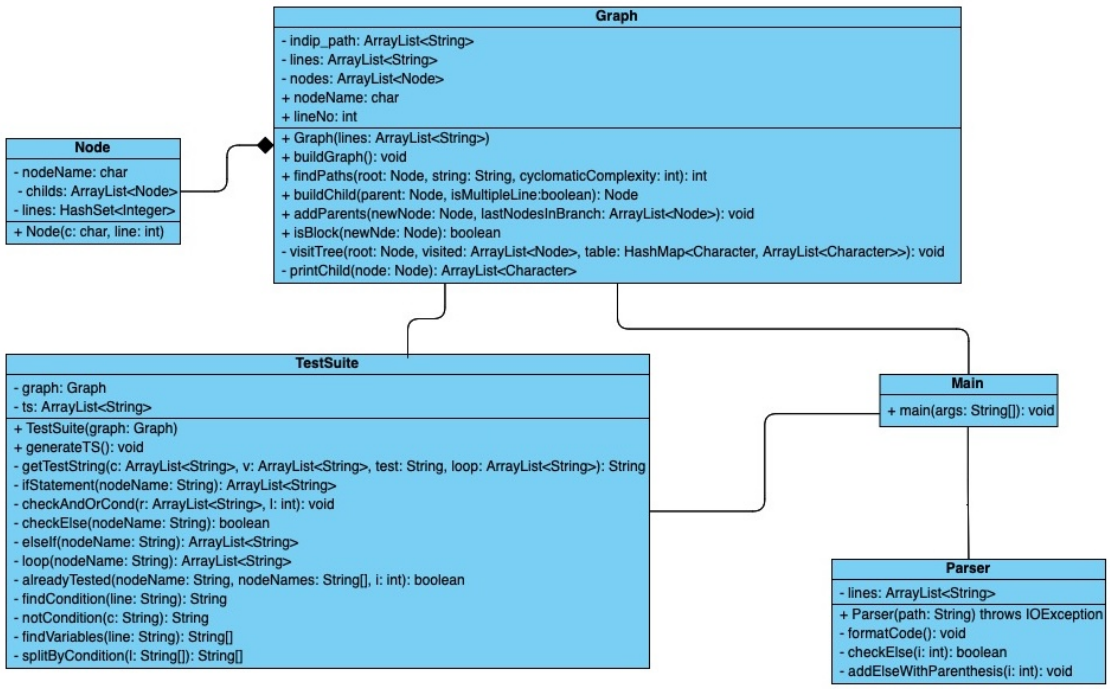

Figura 1: UML Class Diagam

3.1 Parser

La classe Parser svolge un ruolo cruciale nella preparazione del codice sorgente per la costruzione del Control Flow Graph (CFG). Una delle regole fondamentali per la costruzione di un CFG coerente è che ogni istruzione condizionale if debba essere accompagnata da un'istruzione else, anche quando il blocco else non contiene alcuna operazione.

Questa regola è importante perché garantisce che il CFG rifletta correttamente la struttura logica del codice sorgente. Senza un blocco else, il flusso di controllo potrebbe non essere adeguatamente rappresentato, portando a possibili errori nell'analisi del programma.

Il metodo *formatCode()* della classe Parser si occupa di applicare questa regola al codice sorgente. Se trova un'istruzione if senza un corrispondente blocco else, aggiunge automaticamente un blocco else vuoto, garantendo così che il CFG rispetti le regole di struttura e che ogni percorso di esecuzione sia correttamente rappresentato. Utilizza i due metodi ausiliari checkElse(int i) e addElseWithParenthesis(int i) per garantire la corretta formattazione del codice sorgente:

1. Il metodo *checkElse(int i)* esamina le righe del codice sorgente a partire dall'indice i per determinare se esiste un'istruzione else successiva all'istruzione if corrente. Se un blocco else è già presente nel codice, questo metodo restituisce true; altrimenti, restituisce false.

2. Il metodo *addElseWithParenthesis(int i)* cerca la chiusura del blocco if (ovvero una parentesi graffa di chiusura) a partire dall'indice i. Una volta individuata, aggiunge un blocco else subito dopo il blocco if e successivamente aggiunge una riga di codice, come ad esempio printf();, per garantire che il blocco else non sia vuoto.

```
1 usage
private void formatCode() {
    for(int i=0; i<lines.size(); i++) {
        if(lines.get(i).contains("if") && !lines.get(i).contains("else if")) {
            if(!checkElse(i)) {
                if(lines.get(i).contains("{")) {
                    addElseWithParenthesis(i);
                } else {
                    lines.add( index: i+2,  element: "else");
                    lines.add( index: i+3,  element: "printf();");
                }
            }
        }
    }
}
```

Figura 2: Metodo formatCode() della classe Parser

3.2 Node

La classe Node è un componente fondamentale nella costruzione del Control Flow Graph (CFG). Essa rappresenta un singolo nodo all'interno del grafo e memorizza le informazioni relative a quel nodo, inclusi i suoi figli (o nodi successivi nel flusso del programma) e le linee di codice associate a quel nodo.

```
36 usages
public class Node {
    13 usages
    public char nodeName;
    16 usages
    public ArrayList<Node> childs = new ArrayList<>();
    8 usages
    public HashSet<Integer> lines = new HashSet<Integer>();

    7 usages
    public Node(char c, int line) {
        this.nodeName = c;
        this.lines.add(line);
    }
}
```

Figura 3: Classe Node

3.3 Graph

La classe Graph ha il compito di prendere in input il codice sorgente e costruire il Control Flow Graph (CFG). Inoltre, fornisce la funzionalità per individuare i percorsi indipendenti, che rappresentano tutti i possibili percorsi attraverso l'albero del CFG. La classe è formata da molti metodi, ma i due principali sono *buildChild(Node parent, boolean isMultipleLine)* e *findPaths(Node root, String string, int cyclomaticComplexity)*.

1. *buildChild(Node parent, boolean isMultipleLine)*: questo metodo è responsabile della costruzione dei nodi dell'albero del CFG, a partire dal nodo genitore specificato. Durante l'esecuzione, il metodo analizza le righe del codice sorgente, identifica le strutture di controllo come if, else if, else, for, while e do, nonché le istruzioni semplici. Per ciascuna di queste istruzioni, crea un nuovo nodo e lo collega al nodo genitore corrispondente. Utilizzando la ricorsione, gestisce in modo efficace i casi di annidamento delle strutture di controllo. Inoltre, il metodo tiene traccia dei nodi e delle linee di codice durante il processo di costruzione.

2. *findPaths(Node root, String string, int cyclomaticComplexity)*: questo metodo trova tutti i percorsi indipendenti attraverso l'albero del CFG. Utilizzando un approccio ricorsivo, il metodo esplora tutti i possibili percorsi attraverso l'albero del CFG. Durante questa ricerca, tiene traccia delle istruzioni visitate e dei percorsi parziali attraverso l'albero. Utilizzando un'approccio basato sulla profondità, individua i percorsi che coprono tutti i nodi del CFG senza formare cicli. I percorsi indipendenti trovati vengono memorizzati per l'analisi successiva, consentendo di identificare le aree critiche del programma e valutare la sua complessità ciclomatica.

3.3.1 Esempi

Per chiarire ulteriormente il funzionamento della classe Graph, esploreremo degli esempi pratici che illustrano come vengono costruiti i nodi del CFG e come vengono individuati i percorsi indipendenti all'interno del CFG.

1. Supponiamo di avere il seguente codice sorgente in linguaggio C:

```c
if (x > 0)
    printf("x is positive");
else
    printf("x is non-positive");
```

Il metodo *buildChild()* analizza questo codice e crea i nodi corrispondenti nel CFG. Il grafo in output è il seguente:

```
The graph:
A-->(B)
A[0]
B-->(C,D)
B[1]
C-->(F)
C[2]
F-->()
F[5]
D-->(E)
D[3]
E-->(F)
E[4]
```

Figura 4: CFG Esempio 1

Nel Control Flow Graph (CFG) generato per il codice fornito, il nodo A rappresenta l'inizio del programma, corrispondente alla linea zero del codice sorgente. Il nodo B è associato all'istruzione if della linea 1 del codice. Tale nodo ha due figli distinti: il nodo C, che rappresenta l'istruzione eseguita quando la condizione $x > 0$ è vera, e il nodo D, relativo all'istruzione else. Proseguendo, notiamo che il nodo C ha come figlio il nodo F, che segna il termine del blocco di codice condizionale. D'altra parte, il nodo D funge da genitore per il nodo E, che rappresenta l'istruzione situata alla quarta linea del codice.
I percorsi indipendenti trovati sono i seguenti:

```
Indepent paths:
A-->B-->C-->F
A-->B-->D-->E-->F
```

Figura 5: Percorsi indipendenti Esempio 1

2. Supponiamo di avere il seguente codice sorgente in linguaggio C:

```c
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0)
        printf("%d is even", i);
   else
        printf("%d is odd", i);
}
```

Il CFG generato è il seguente:

```
The graph:
A-->(B)
A[0]
B-->(C,H)
B[1]
C-->(D,E)
C[2]
D-->(G)
D[3]
G-->(B)
G[6]
E-->(F)
E[4]
F-->(G)
F[5]
H-->()
H[7]
```

Figura 6: CFG Esempio 2

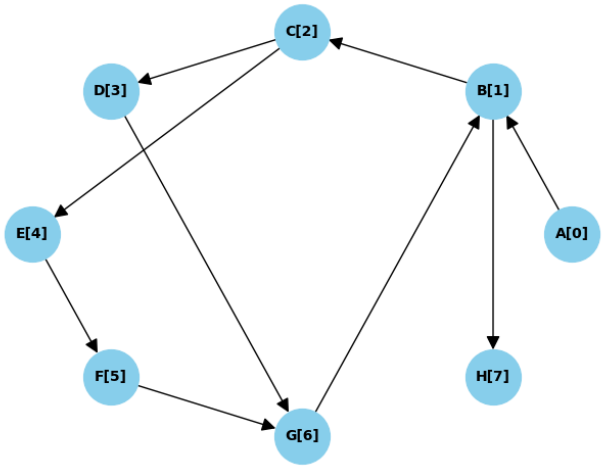

Figura 7: Visualizzazione del CFG

Nel CFG, sono identificabili tre percorsi indipendenti, ognuno dei quali rappresenta un diverso percorso di esecuzione all'interno del programma. Il primo percorso attraversa il ciclo "for" e soddisfa la condizione dell'istruzione "if". In altre parole, il programma esegue l'iterazione del ciclo "for" e, all'interno di ciascuna iterazione, verifica se la variabile "i" soddisfa la condizione per essere pari. Se questa condizione è verificata, il programma esegue l'istruzione all'interno del blocco "if". Il secondo percorso segue lo stesso ciclo "for", ma al contrario del primo, non soddisfa la condizione dell'istruzione "if". Di conseguenza, il programma esegue l'istruzione alternativa specificata nel blocco "else". Infine, il terzo percorso non coinvolge il ciclo "for". Questo percorso rappresenta il caso in cui il programma non entra affatto nel ciclo "for", magari perché la condizione iniziale del ciclo non è soddisfatta.

```
Indepent paths:
A-->B-->C-->D-->G-->B-->H
A-->B-->C-->E-->F-->G-->B-->H
A-->B-->H
```

Figura 8: Percorsi indipendenti Esempio 2

3.4 TestSuite

La classe TestSuite è responsabile della generazione della suite di test a partire dal codice sorgente e dai percorsi indipendenti individuati nel Control Flow Graph (CFG).

Nel metodo *generateTS()*, vengono inizializzate delle variabili utili e viene iterato attraverso ciascun percorso indipendente presente nella lista *indipPath*. Per ogni percorso, vengono analizzati i nomi dei nodi che lo compongono, suddivisi in base al separatore "−− >".

All'interno del ciclo, vengono eseguite diverse operazioni:

- Vengono individuate e analizzate le istruzioni di tipo "if" tramite il metodo *ifStatement()*, che restituisce le condizioni e le variabili coinvolte negli statement condizionali.
- Vengono considerati gli statement "else if", dove presenti, attraverso il metodo *elseIf()*.
- Si gestiscono eventuali cicli "for" o "while" tramite il metodo *loop()*.
- Si verifica se è presente un blocco "else" tramite il metodo *checkElse()*.

In base alle condizioni e alle istruzioni presenti nei nodi del percorso, vengono costruiti i test corrispondenti, tenendo conto delle condizioni e delle variabili coinvolte.

I metodi ausiliari della classe, come *getTestString(), checkAndOrCond(), findCondition(), notCondition(), findVariables() e splitByCondition()* supportano il processo di generazione dei test fornendo funzionalità per l'analisi e la manipolazione delle condizioni e delle istruzioni presenti nei nodi del CFG.

Di seguito vengono presentati i test generati per gli esempi di percorsi indipendenti precedentemente discussi, ottenuti mediante l'esecuzione della classe TestSuite.

1. Ripresentiamo il codice del primo esempio.

```
1 if (x > 0)
2     printf("x is positive");
3 else
4     printf("x is non-positive");
```

   La semplicità della Test Suite in questo contesto deriva dalla presenza di una singola variabile nel codice sorgente. Di seguito viene riportato l'output prodotto dalla classe TestSuite:

```
The Test Suite:
0. x  >  0;
1. x  <=  0;
```

Figura 9: Test Suite Esempio 1

2. Rivediamo l'esempio 2.

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0)
        printf("%d is even", i);
   else
        printf("%d is odd", i);
}
```

Per la Test Suite, essa è strutturata in modo semplice e comprende tre test set, ciascuno dei quali segue uno dei tre percorsi indipendenti identificati nel CFG, come mostrato in Figura 10. I test set sono stati progettati per coprire e testare i vari scenari di esecuzione previsti dai percorsi del grafo.

```
The Test Suite:
0.  i  <  10; i % 2  ==  0;
1.  i  <  10; i % 2  !=  0;
2.  i  >=  10;
```

Figura 10: Test Suite Esempio 2

4 Risultati

Nell'ambito di questa analisi, il programma è stato applicato a tre algoritmi di ordinamento implementati in linguaggio C: il Bubblesort, il Selection Sort e il Quicksort. L'obiettivo è stato quello di esaminare il CFG di ciascun algoritmo, identificare i percorsi indipendenti all'interno di esso e generare una Test Suite corrispondente. Mediante questo processo, è possibile comprendere la struttura e il comportamento di tali algoritmi durante l'esecuzione, nonché valutare l'efficacia della suite di test generata nel garantire una copertura completa delle varie situazioni e condizioni di esecuzione.

4.1 Bubble Sort

Ricordiamo l'algoritmo Bubble Sort

```
int iter, count;
boolean swap_found;
for (count=0; count<N; count++) {
   Perm[count] = count;
}
iter = 0;
do {
    swap_found=FALSE;
    for (count=0; count < N-iter-1; count++) {
        if (V[Perm[count]] > V[Perm[count+1]]) {
            swap(Perm, count, count+1);
            swap_found = TRUE;
        }
    }
    iter ++;
}while(swap_found==TRUE);
```

Se andiamo a vedere il CFG abbiamo un grafo molto più complesso, che diventa difficile leggere anche graficamente.

I percorsi indipendenti risultati dal CFG sono:

```
The graph:
A-->(B)
A[0]
B-->(C)
B[1, 2]
C-->(D,E)
C[3]
D-->(C)
D[4, 5]
E-->(F)
E[6]
F-->(G)
F[7]
G-->(H)
G[8]
H-->(I,N)
H[9]
I-->(J,K)
I[10]
J-->(M)
J[11, 12, 13]
M-->(H)
M[16]
K-->(L)
K[14]
L-->(M)
L[15]
N-->(F,O)
N[17, 18]
O-->()
O[19]
```

Figura 11: CFG Bubble Sort

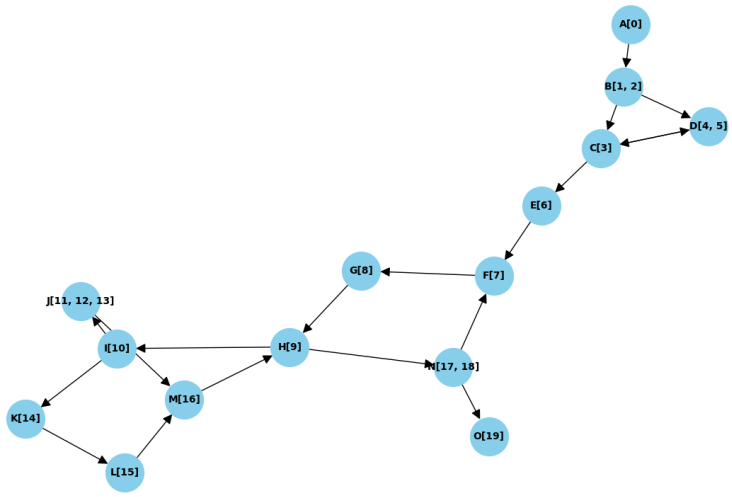

Figura 12: Visualizzazione del CFG

1. $A \to B \to C \to D \to C \to E \to F \to G \to H \to I \to J \to M \to H \to N \to O$
2. $A \to B \to C \to D \to C \to E \to F \to G \to H \to I \to K \to L \to M \to H \to N \to O$
3. $A \to B \to C \to D \to C \to E \to F \to G \to H \to N \to O$
4. $A \to B \to C \to E \to F \to G \to H \to I \to J \to M \to H \to N \to O$
5. $A \to B \to C \to E \to F \to G \to H \to I \to K \to L \to M \to H \to N \to O$
6. $A \to B \to C \to E \to F \to G \to H \to N \to O$

La Test Suite per l'algoritmo di ordinamento Bubble Sort è:

1. count $<$ N; count $<$ N-iter-1; $V[Perm[count]] > V[Perm[count+1]]$; $swap\_found == TRUE$;
2. count $<$ N; count $<$ N-iter-1; $V[Perm[count]] \leq V[Perm[count+1]]$; $swap\_found \neq TRUE$; $swap\_found == TRUE$;
3. count $<$ N; count $\geq N - iter - 1$; $swap\_found == TRUE$;
4. count $\geq N$; count $<$ N-iter-1; $V[Perm[count]] > V[Perm[count+1]]$; $swap\_found == TRUE$;
5. count $\geq N$; count $<$ N-iter-1; count $\geq N$; $V[Perm[count]] \leq V[Perm[count+1]]$; $swap\_found \neq TRUE$; count $\geq N$; $swap\_found \neq TRUE$; count $\geq N$; $swap\_found \neq TRUE$; $V[Perm[count]] \leq V[Perm[count+1]]$; $swap\_found \neq TRUE$; $swap\_found == TRUE$;
6. count $\geq N$; count $\geq N - iter - 1$; $swap\_found == TRUE$;

4.2 Selection Sort

Ricordiamo l'algoritmo Selection Sort

```
1     int iter, count, count_of_max;
2     for ( count=0; count<N; count++ ) {
3        Perm[count] = count;
4     }
5
6     for ( iter=0; iter<N-1; iter++ ) {
7         for (count=1, count_of_max=0; count<N-iter; count++) {
8             if (V[Perm[count]] > V[Perm[count_of_max]])
9                 count_of_max = count;
10        }
11        swap(Perm,count_of_max,N-iter-1);
12    }
```

Diamo ora uno sguardo ai risultati ottenuti dal programma, che includono il CFG, i percorsi indipendenti identificati e la Test Suite generata.

```
The graph:
A-->(B)
A[0]
B-->(C)
B[1]
C-->(D,E)
C[2]
D-->(C)
D[3, 4]
E-->(F,M)
E[6]
F-->(G,L)
F[7]
G-->(H,I)
G[8]
H-->(K)
H[9]
K-->(F)
K[12]
I-->(J)
I[10]
J-->(K)
J[11]
L-->(E)
L[13, 14]
M-->()
M[15]
```

Figura 13: CFG Selection Sort

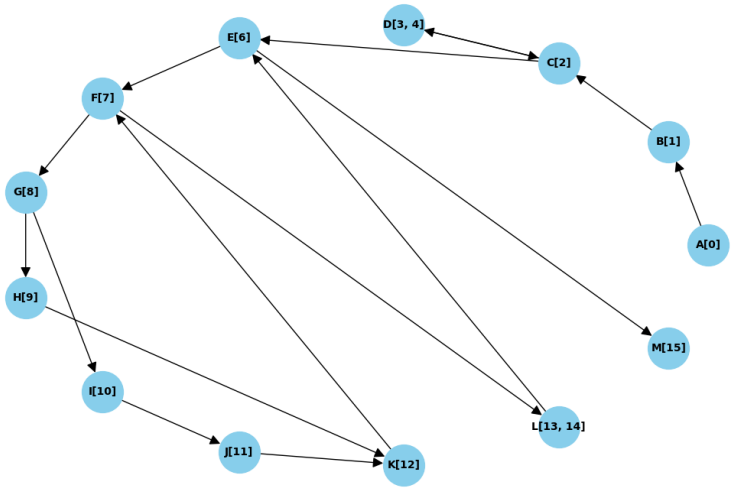

Figura 14: Visualizzazione del CFG

I percorsi indipendenti risultati dal CFG sono:

```
Independent paths:
A-->B-->C-->D-->C-->E-->F-->G-->H-->K-->F-->L-->E-->M
A-->B-->C-->D-->C-->E-->F-->G-->I-->J-->K-->F-->L-->E-->M
A-->B-->C-->D-->C-->E-->F-->L-->E-->M
A-->B-->C-->D-->C-->E-->M
A-->B-->C-->E-->F-->G-->H-->K-->F-->L-->E-->M
A-->B-->C-->E-->F-->G-->I-->J-->K-->F-->L-->E-->M
A-->B-->C-->E-->F-->L-->E-->M
A-->B-->C-->E-->M
```

Figura 15: Percorsi indipendenti Selection Sort

La Test Suite per l'algoritmo di ordinamento Selection Sort è:

```
The Test Suite:
0.  count < N;  iter < N-1;  count < N-iter; V[Perm[count]]  >  V[Perm[count_of_max]];
1.  count < N;  iter < N-1;  count < N-iter; V[Perm[count]]  <=  V[Perm[count_of_max]];
2.  count < N;  iter < N-1;  count >= N-iter;
3.  count < N;  iter >= N-1;
4.  count >= N;  iter < N-1;  count < N-iter; V[Perm[count]]  >  V[Perm[count_of_max]];
5.  count >= N;  iter < N-1;  count < N-iter; V[Perm[count]]  <=  V[Perm[count_of_max]]; V[Perm[count]]  <=  V[Perm[count_of_max]];
6.  count >= N;  iter < N-1;  count >= N-iter;
7.  count >= N;  iter >= N-1;
```

Figura 16: Test Suite Selection Sort

4.3 Quick Sort

Ricordiamo l'algoritmo Quick Sort

```
1     int l, r, pivot;
2     pivot = V[0];
3     l = 0;
4     r = N;
5     while ( l < r ) {
6           do {
7               r--;
8           }while ( V[r] > pivot && r > l );
9           if ( r != l ) {
10                do {
11                    l++;
12                } while ( V[l] <= pivot && l < r );
13                swap(V, l, r);
14          }
15    }
16    swap(V, l, 0);
```

Diamo ora uno sguardo ai risultati ottenuti dal programma, che includono il CFG, i percorsi indipendenti identificati e la Test Suite generata.

```
The graph:
A-->(B)
A[0]
B-->(C)
B[1, 2, 3, 4]
C-->(D,M)
C[5]
D-->(E)
D[6]
E-->(D,F)
E[7, 8]
F-->(G,J)
F[9]
G-->(H)
G[10]
H-->(G,I)
H[11, 12]
I-->(L)
I[13, 14]
L-->(C)
L[17]
J-->(K)
J[15]
K-->(L)
K[16]
M-->()
M[18, 19]
```

Figura 17: CFG Quick Sort

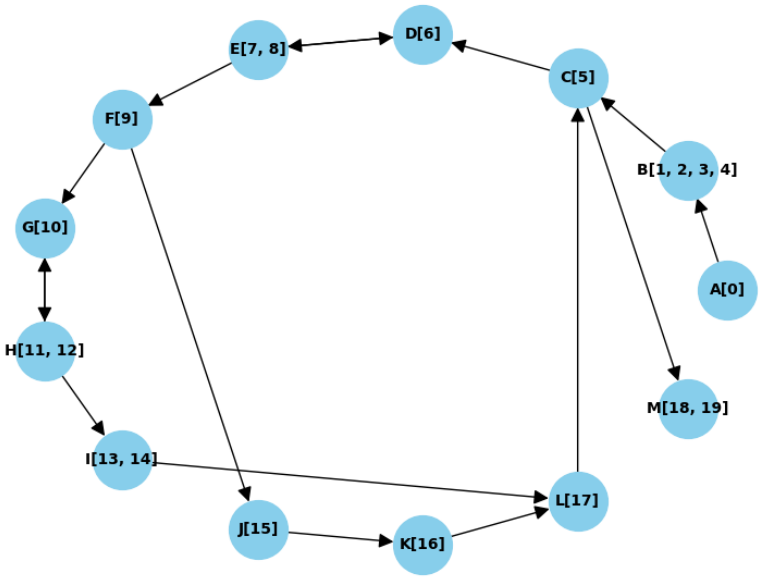

Figura 18: Visualizzazione del CFG

I percorsi indipendenti risultati dal CFG sono:

```
Independent paths:
A-->B-->C-->D-->E-->F-->G-->H-->I-->L-->C-->M
A-->B-->C-->D-->E-->F-->J-->K-->L-->C-->M
A-->B-->C-->M
```

Figura 19: Percorsi indipendenti Quick Sort

La Test Suite per l'algoritmo di ordinamento Quick Sort è:

```
The Test Suite:
0. l < r ; V[r] > pivot ; r > l ; r != l ; V[l] < pivot ; l < r ;
1. l < r ; V[r] > pivot ; r > l ; r == l ; V[l] >= pivot ;
2. l >= r ;
```

Figura 20: Test Suite Quick Sort