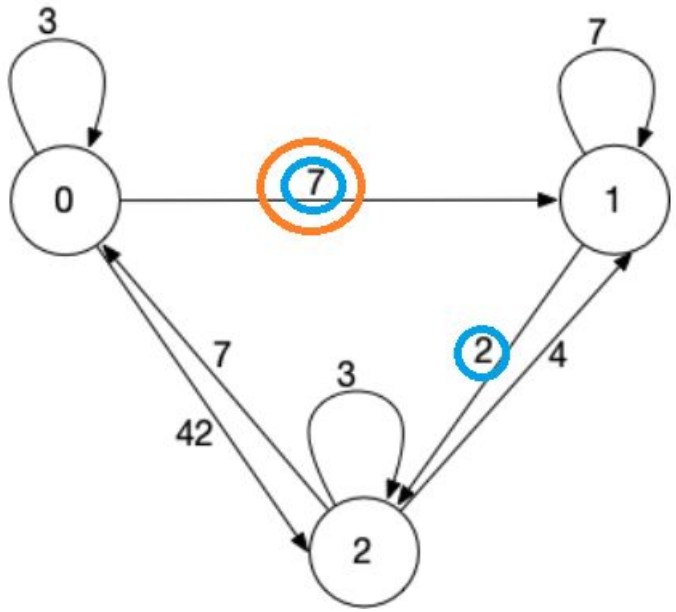
Sintesi delle specifiche

L'input si compone di due numeri, il numero di nodi dei grafi e la lunghezza della classifica (k), seguiti da una serie di comandi, di due soli tipi:

1. AggiungiGrafo [matrice-di-adiacenza]
2. TopK

L'obiettivo è stampare ad ogni comando TopK i top-k grafi, in ordine crescente per somma dei cammini più brevi tra il nodo 0 e tutti gli altri nodi del grafo raggiungibili da 0.

```
AggiungiGrafo
3,7,42
0,7,2
7,4,3
```



Da 0 a 1: 7
Da 0 a 2: 7 + 2 = 9
Totale = 16

Ambiente di sviluppo e compilazione

Ambiente di sviluppo consigliato: **linux**

Compilatore: **gcc**

Possible usare altri IDE

Compilare il programma su terminale:

```
$gcc -Wall -Werror -O2 -g3 sorgente.c -o eseguibile
```

-Wall: mostra tutti i warnings

-Werror: tutti i warnings sono trattati come errori

-O2: abilita alcune ottimizzazioni del codice da parte del compilatore

-g3: include le informazioni utili per il debugger

Il verificatore utilizza anche i flag `-static` e `-s` , ma potrebbero darvi problemi in locale, specialmente su MacOS.

Il verificatore include anche il flag `-DEVAL` che definisce automaticamente la macro `EVAL`, è l'equivalente di inserire nel codice `#define EVAL` (vedi slide successive)

Comandi utili (linux) - 1

Fornire contenuto del file in input al programma

```
$./programma < public_input_file
```

Scrivere output del programma su file

```
$./programma < public_input_file > program_output
```

Comandi utili (linux) -2

Confrontare contenuto di due file

```
$ diff ./public_output ./program_output
```

(nessun output significa che i due file sono identici)

Nelle specifiche di quest'anno non è però richiesto che l'output sia ordinato, di conseguenza ci possono essere molteplici output diversi ma corretti per un dato input.

Gli output forniti con i test pubblici sono comunque ordinati crescenti, quindi per confrontare con quelli potete inserire l'ordinamento nel vostro codice. Per evitare che questo appesantisca inutilmente l'esecuzione disabilitatelo quando testate sul verificatore.

Per fare questo in modo comodo potete sfruttare la macro `EVAL` definita dal verificatore in fase di compilazione. Se non è presente fate l'ordinamento dell'output, altrimenti non fatelo:

```
#ifndef  EVAL

ordina()

#endif
```

Errori verificatore

- **Output is not correct**:
  Causato da errata implementazione, errato parsing dell'input o errata scrittura dell'output.
  Testare in locale con test pubblico.

- **Execution timed out – execution killed**:
  Implementazione non soddisfa i vincoli di tempo e memoria del test.
  Utilizzare valgrind per identificare le parti di codice più dispendiose o memory leaks; valutare strutture dati appropriate.

- **Execution killed with signal 11**:
  Implementazione non soddisfa vincoli di memoria o si è verificato segmentation fault. Utilizzare valgrind in locale.

- **Execution failed because the return code was nonzero**:
  Il main non ritorna 0 o il programma termina inaspettatamente.
  Debug locale.

Generatore di Test

Tool in python per generare nuovi test.

NB: è necessario aver installato una versione di Python >= 3

Ha una serie di parametri, alcuni obbligatori, altri opzionali.

```
$ python inputgen.py test generato.txt 10 3 50
--topk_start --topk_end --topk_every 2
```

Oppure:

```
$ python3 inputgen.py test generato.txt 10 3 50
--topk_start --topk_end --topk_every 2
```

Generatore di Test - parametri

filename, **Nome del file da generare**

d, type=int, **Numero di nodi dei grafi da generare**

k, type=int, **Lunghezza della classifica**

size, type=int, **Numero di grafi da generare**

--edge_prob, type=float, default=0.5, **Probabilità con cui inserire un arco nel grafo**

--weight_min, type=int, default=0, **Valore minimo dei pesi dei grafi**

--weight_max type=int, default=2**32-1 **Valore massimo dei pesi dei grafi**

--decreasing **Rendi i pesi degli archi tendenzialmente decrescenti**

--topk_start **Aggiungi comando TopK all'inizio**

--topk_end, **Aggiungi TopK alla fine**

--topk_every, type=int, **Aggiungi TopK ogni TOPK_EVERY matrici**

--topk_prob, type=float, **Aggiungi TopK con una certa probabilità**

Installare Python (OS X)

1. Installare Homebrew con il seguente comando

```
$ /bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/instal
l/master/install.sh)"
```

2. Installare Python con il seguente comando

```
$ brew install python
```

Installare Python (Windows)

https://www.python.org/downloads/windows/

Valgrind - memcheck

Per installarlo (Ubuntu):
`$ sudo apt-get install valgrind`

Memcheck è il tool principale, consente di rilevare errori nell'uso della memoria dinamica, come accessi ad aree non allocate, e i leaks di memoria, ovvero situazioni in cui non viene liberata memoria precedentemente allocata e non più referenziata.

Per utilizzarlo :

1) Compilare il sorgente con il flag **-g3**
2) `$ valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./eseguibile`

Valgrind - callgrind

Callgrind / kcachegrind

Callgrind è un altro tool di Valgrind che consente di visualizzare il tempo impiegato nelle diverse funzioni/istruzioni del programma

1) Per installare la GUI (interfaccia grafica):
   `$ sudo apt-get install kcachegrind`

2) Per generare report testuale:
   `$ valgrind --tool=callgrind --callgrind-out-file=outputfile ./eseguibile`

   Per analizzarlo in kcachegrind:
3) `$ kcachegrind outputfile`

Valgrind - massif

Massif / massif-visualizer

Mass-If è un altro tool di valgrind che consente di visualizzare l'andamento della memoria allocata nel corso dell'esecuzione

1) Per installare la GUI (interfaccia grafica):
   `$ sudo apt-get install massif-visualizer`

2) Per generare report testuale:
   `$ valgrind --tool=massif --massif-out-file=outputfile ./eseguibile`

   Per visualizzarlo in massif-visualizer:
3) `$ massif-visualizer outputfile`

AddressSanitizer - 1

AddressSanitizer (aka ASan) è un memory error detector per C/C++.
Per utilizzarlo bisogna compilare il programma usando gcc con il flag **-fsanitize=address** (oltre a quelli riportati sopra).

Esempio use-after-free:

```
#include <stdlib.h>
int main() {
  char *x = (char*)malloc(10 * sizeof(char*));
  free(x);
  return x[5];
 }
```

AddressSanitizer - 2

AddressSanitizer (aka ASan) è un memory error detector per C/C++.
Per utilizzarlo bisogna compilare il programma usando gcc con il flag **-fsanitize=address** (oltre a quelli riportati sopra).

Output:

```
==9901==ERROR: AddressSanitizer: heap-use-after-free on address 0x60700000dfb5 at pc
0x45917b bp 0x7fff4490c700 sp 0x7fff4490c6f8
READ of size 1 at 0x60700000dfb5 thread T0
    #0 0x45917a in main use-after-free.c:5
    #1 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226
    #2 0x459074 in _start (a.out+0x459074)
0x60700000dfb5 is located 5 bytes inside of 80-byte region
[0x60700000dfb0,0x60700000e000)
freed by thread T0 here:
    #0 0x4441ee in __interceptor_free
projects/compiler-rt/lib/asan/asan malloc linux.cc:64
    #1 0x45914a in main use-after-free.c:4
    #2 0x7fce9f25e76c in   libc start main /build/buildd/eglibc-2.15/csu/libc-start.c:226
previously allocated by thread T0 here:
    #0 0x44436e in __interceptor_malloc
projects/compiler-rt/lib/asan/asan malloc linux.cc:74
    #1 0x45913f in main use-after-free.c:3
    #2 0x7fce9f25e76c in   libc start main /build/buildd/eglibc-2.15/csu/libc-start.c:226
  SUMMARY: AddressSanitizer: heap-use-after-free use-after-free.c:5 main
```