Corso di laurea: *Informatica*
Anno accademico: 2013/2014

Insegnamento: Tecniche di Simulazione
Prof.ssa Francesca Mazzia

Caso di studio:

“Studio delle performance di un sistema di smistamento di lavori”

Autori:
Alessandro Suglia - matricola 587176
Gianvito Taneburgo - matricola 587645

Indice

Formulazione ed analisi del problema

Il caso di studio assegnato per condurre la sperimentazione di un processo di simulazione è tratto dalla quarta edizione del libro *Discrete-Event System Simulation* (Banks, Carson, Nelson, Nicol – Pearson Prentice-Hall).

Di seguito vengono riportate le tracce dei due esercizi svolti, tratte dal suddetto libro.

Chapter 11, exercise 13

Jobs enter a job shop in random fashion according to a Poisson process at a stationary overall rate, two every 8-hour day. The jobs are of four types. They flow from work station to work station in a fixed order, depending on type, as shown next. The proportions of each type are also shown.

Type	Flow through Stations	Proportion
1	1, 2, 3, 4	0.4
2	1, 3, 4	0.3
3	2, 4, 3	0.2
4	1, 4	0.1

Processing times per job at each station depend on type, but all times are (approximately) normally distributed with mean and s.d. (in hours) as follows:

	Station			
Type	1	2	3	4
1	(20, 3)	(30, 5)	(75, 4)	(20, 3)
2	(18, 2)		(60, 5)	(10, 1)
3		(20, 2)	(50, 8)	(10, 1)
4	(30, 5)			(15, 2)

Station i will have c_i workers (i = 1, 2, 3, 4). Each job occupies one worker at a station for the duration of a processing time. All jobs are processed on a first-in-first-out basis, and all queues for waiting jobs are assumed to have unlimited capacity. Simulate the system for 800 hours, preceded by a 200-hour initialization period. Assume that $c_1 = 8$, $c_2 = 8$, $c_3 = 20$, $c_4 = 7$. Based on R = 5 replications, compute a 97.5% confidence interval for average worker utilization at each of the four stations. Also, compute a 95% confidence interval for mean total response time for each job type, where a total response time is the total time that a job spends in the shop.

Chapter 11, exercise 14

Change Exercise 13 to give priority at each station to the jobs by type. Type 1 jobs have priority over type 2, type 2 over type 3, and type 3 over type 4. Use 800 hours as run length, 200 hours as initialization period, and R = 5 replications. Compute four 97.5% confidence intervals for mean total response time by type. Also, run the model without priorities and compute the same confidence intervals. Discuss the trade-offs when using *first in, first out* versus a priority system.

Nella traduzione in italiano è stata adottata la seguente nomenclatura:

job shop = sistema
job = lavoro
job type = tipo di lavoro
worker = lavoratore
(work) station = stazione (di lavoro)
processing time = tempo di servizio

La finalità del primo esercizio è calcolare, per ogni stazione, un *intervallo di confidenza* del 97,5% dell'utilizzo medio dei suoi lavoratori e, per ciascun tipo di lavoro, un *intervallo di confidenza* del 95% del tempo medio di permanenza nel sistema dei lavori di quel tipo.
Il secondo esercizio, similmente al primo, richiede di calcolare, per ciascun tipo di lavoro, un intervallo di confidenza del 97,5% del tempo medio di permanenza nel sistema dei lavori di quel tipo, prima assegnando una priorità ai lavori e dopo senza priorità.
Al termine della simulazione saranno perciò discussi i vantaggi e gli svantaggi di ciascuna politica di gestione delle code (FIFO e per tipo di lavoro).

Da qui in poi con l'espressione “caso di studio” ci si riferirà ad entrambi gli esercizi.

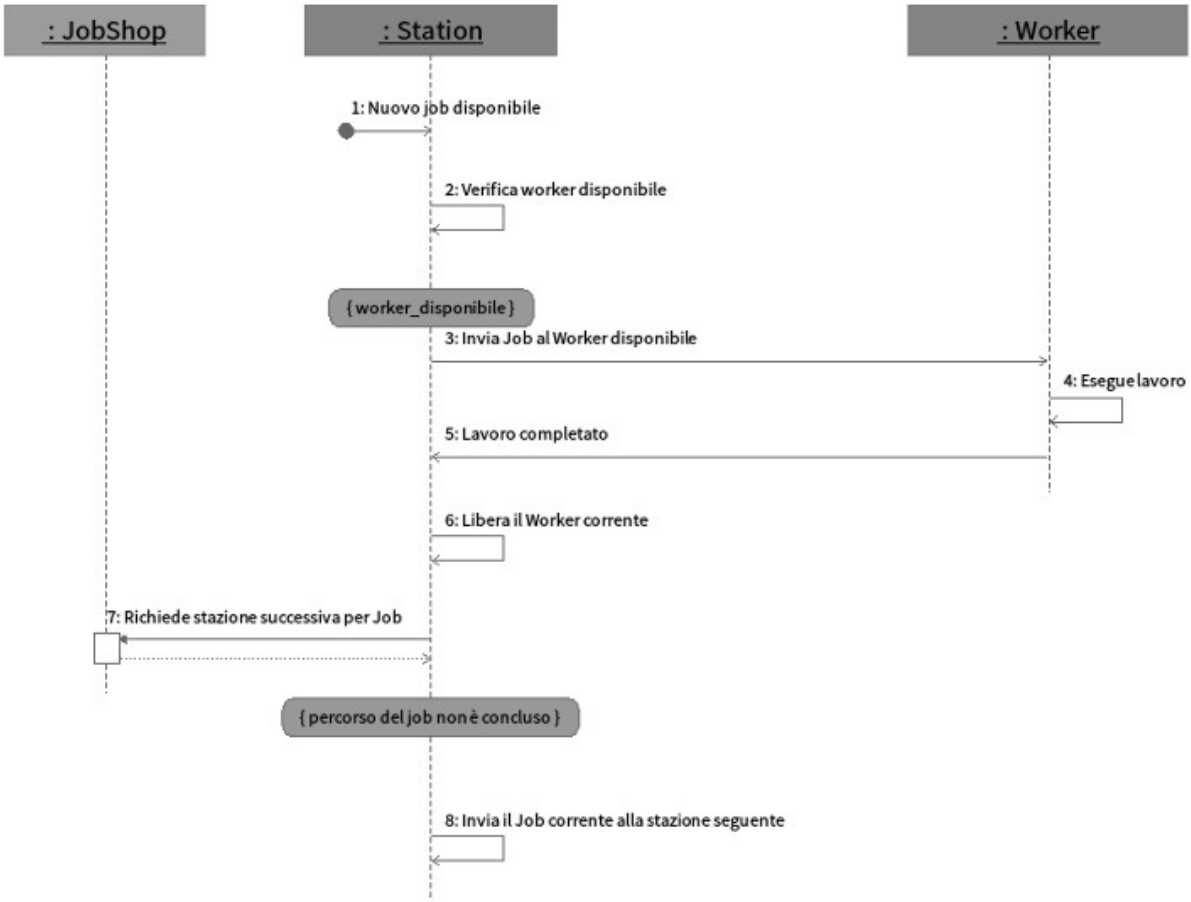
Sviluppo del modello del sistema

La tipologia del sistema da riprodurre e la sua complessità suggeriscono di effettuare la simulazione adottando un approccio per processi. Risulta infatti naturale pensare ad alcuni processi che interagiscono tra loro secondo le dinamiche del mondo reale descritte nel caso di studio.
Il modello progettato prevede:

- un processo che genera nuovi lavori: tale generatore indirizzerà i lavori, in base al loro tipo, alla prima stazione che deve svolgerli;
- un processo per ogni stazione del sistema: la stazione conserverà la lista dei lavori da svolgere e possiederà un predeterminato numero di lavoratori cui assegnarli;
- un processo per ogni lavoratore: svolgerà un lavoro e, qualora necessario, lo indirizzerà alla stazione successiva.

Sarà dunque avviato un numero di processi pari a 1 (generatore clienti) + 4 (stazioni) + 43 (lavoratori) = 48.

L'interazione di questi processi può essere meglio descritta dal seguente diagramma:



L'esercizio 14, che richiede una variante sul precedente, fa emergere la necessità di parametrizzare alcune componenti del sistema, per rendere più agevole il processo di simulazione nel caso di piccole modifiche, evitando una riprogettazione del simulatore.
Un'ispezione del problema condotta tenendo a mente il principio informatico che suggerisce di evitare i “magic numbers” suggerisce di parametrizzare le seguenti componenti:

- distribuzione di probabilità (e suoi parametri) dei tempi di arrivo dei lavori nel sistema;
- numero dei tipi di lavoro;
- proporzioni con cui i tipi vengono assegnati ai nuovi lavori;
- numero di stazioni del sistema;
- percorsi (flusso attraverso le stazioni) di ciascun tipo di lavoro;
- distribuzione di probabilità (e suoi parametri) dei tempi di servizio dei tipi di lavoro per ciascuna stazione;
- numero di lavoratori per ciascuna stazione;
- politica di gestione della coda di lavori;
- tempo totale di simulazione;
- tempo totale di start-up;
- numero di repliche del processo di simulazione;
- percentuali di confidenza degli intervalli di confidenza;
- seme utilizzato per la generazione di numeri pseudo-casuali (per fini soprattutto di debug).

Per tale motivo tutti i parametri appena elencati, insieme ad altri necessari per il funzionamento del simulatore, sono stati raccolti e codificati in un unico file con pubblica visibilità.

Raccolta e/o stima dei dati di input

I dati di input della simulazione non sono stati raccolti dal mondo reale, ma generati dal calcolatore. Il caso di studio, infatti, esplicita già quali sono le distribuzioni di probabilità che scandiscono i tempi della simulazione: risulterebbe pertanto superflua la fase di studio dei dati di input al fine di individuare le distribuzioni che meglio approssimano la realtà.
Ciononostante, i dati generati dal calcolatore sono stati supposti essere provenienti dal mondo reale e su di essi sono stante condotte delle analisi.

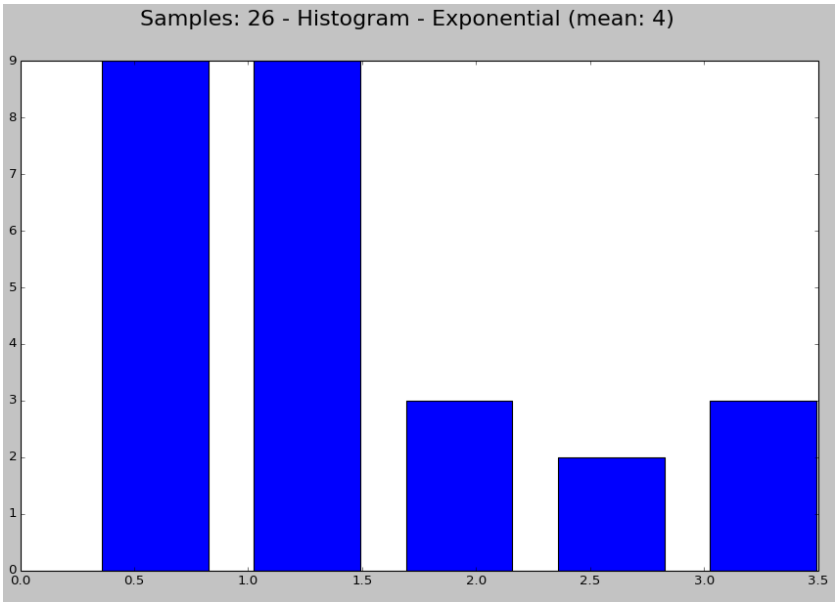
La distribuzione esponenziale e quella normale sono di interesse per lo studio, poiché dettano, rispettivamente, i tempi di interarrivo e quelli di servizio.
Pertanto, coerentemente con le richieste del caso di studio, sono state prese a campione una distribuzione esponenziale di media pari al tasso di arrivo dei lavori nel sistema e una distribuzione normale con media e deviazione standard pari a quelle della prima stazione (per il tipo 1) del sistema.
Per ciascuna distribuzione, sono state condotte queste analisi:

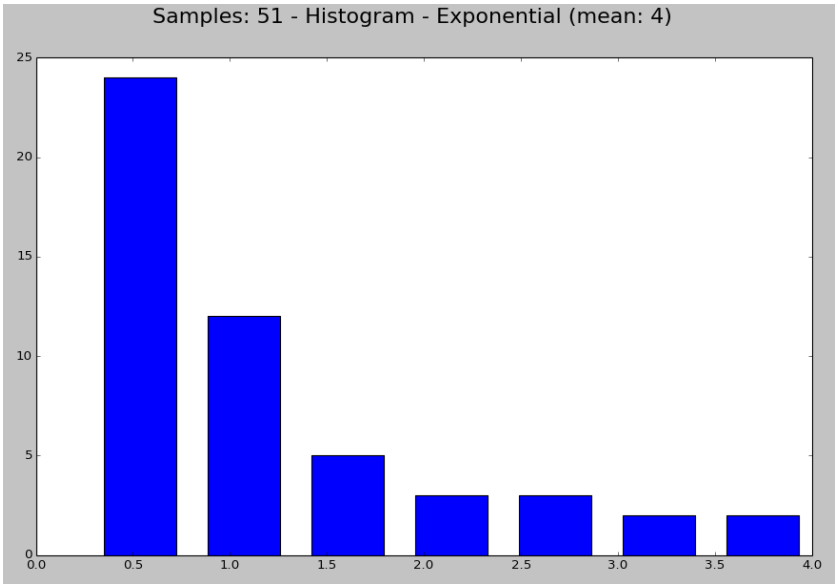
- istogramma per evidenziare le frequenze con cui compaiono i dati;
- grafico quantile-quantile con una distribuzione diversa da quella in esame;
- grafico quantile-quantile con una distribuzione uguale a quella in esame;
- test del chi-quadro con una distribuzione diversa da quella in esame;
- test del chi-quadro con una distribuzione uguale a quella in esame;
- test di Kolmogorov-Smirnov con una distribuzione diversa da quella in esame;
- test di Kolmogorov-Smirnov con una distribuzione uguale a quella in esame.

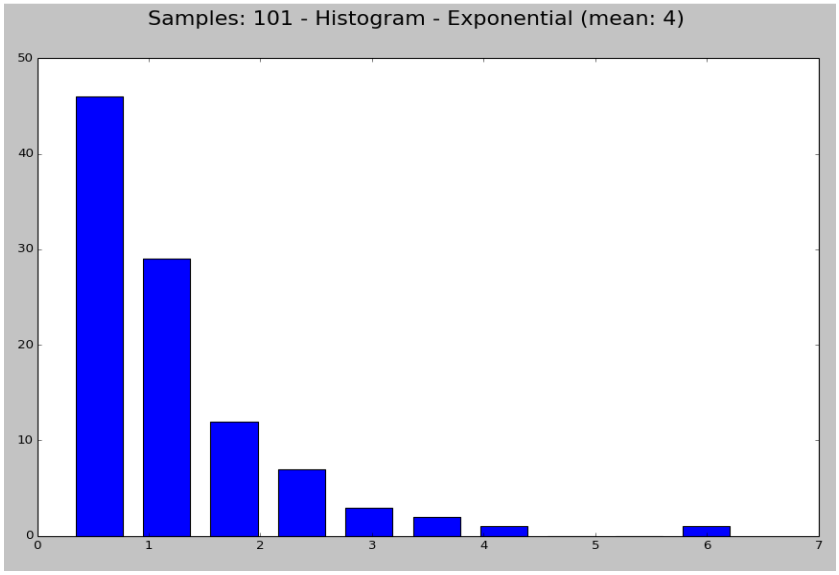
Le analisi sono state ripetute con un numero diverso di campioni in input: 26, 51 e 101. Lo scopo è quello di evidenziare come, con l'aumentare del numero di campioni, la distribuzione di probabilità dei numeri generati tende ad allinearsi sempre più alla distribuzione che li ha generati.

Nella pagine seguenti sono presentati i risultati delle analisi.

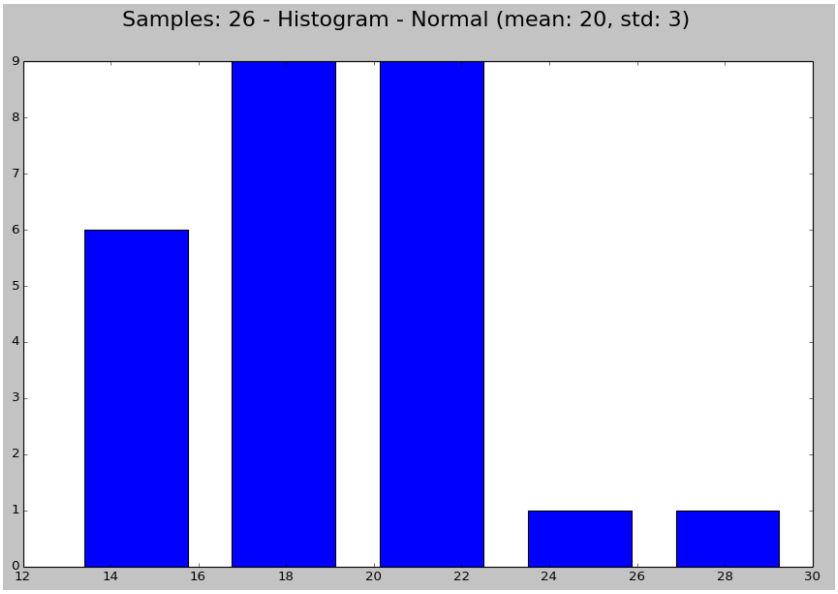
Istogramma dei dati della distribuzione esponenziale

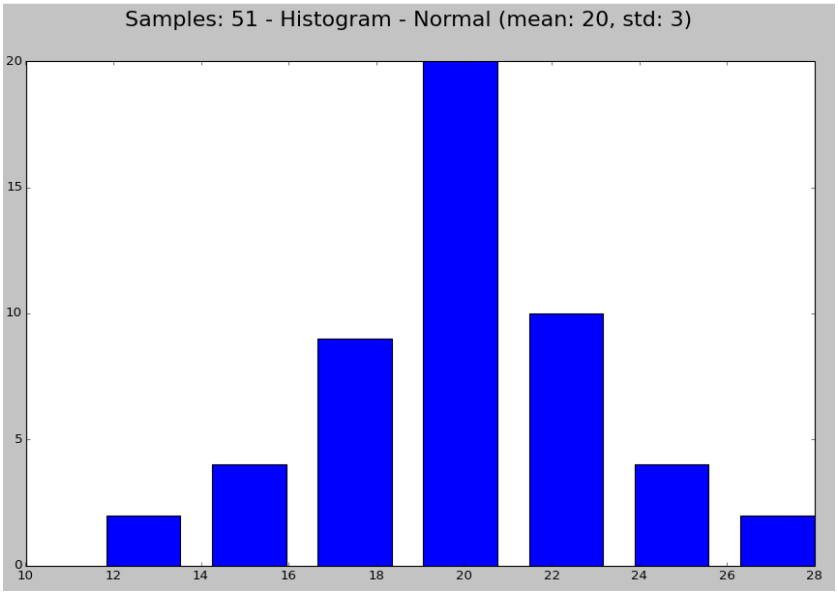






Istogramma dei dati della distribuzione normale





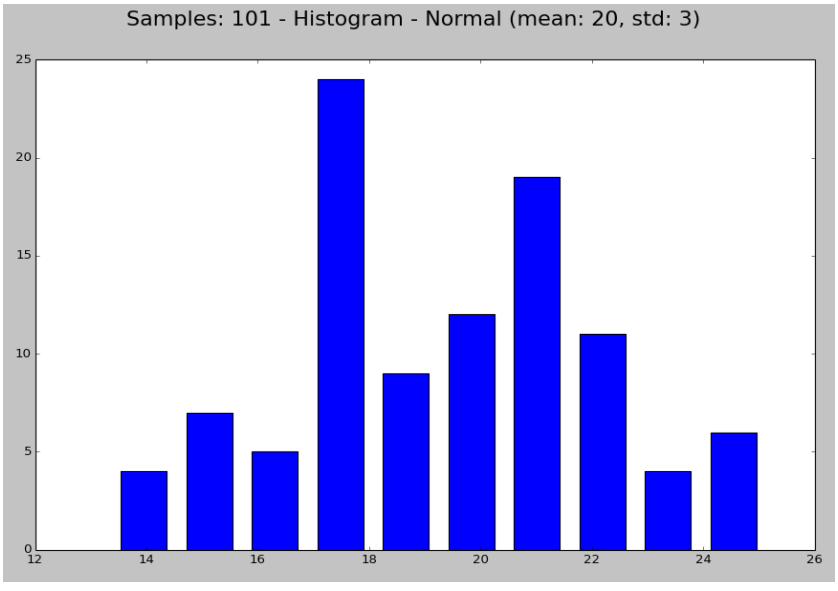
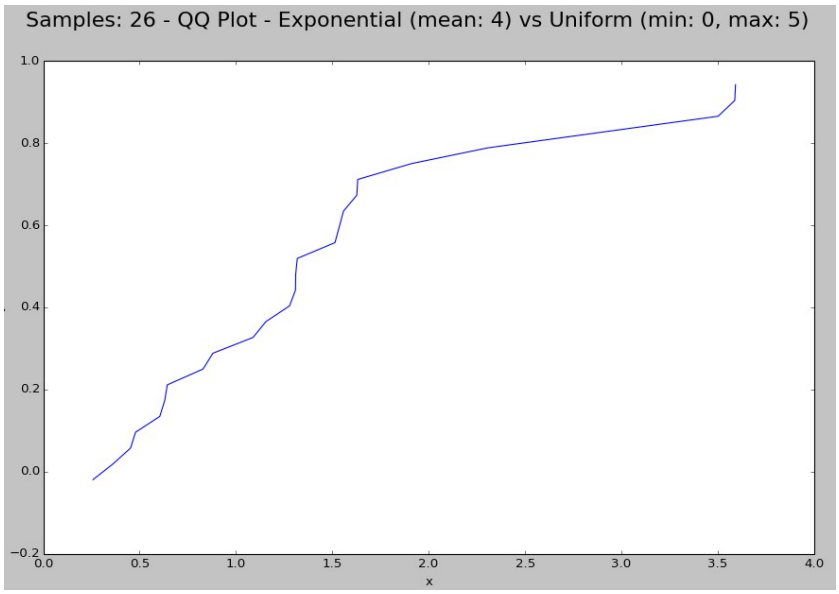
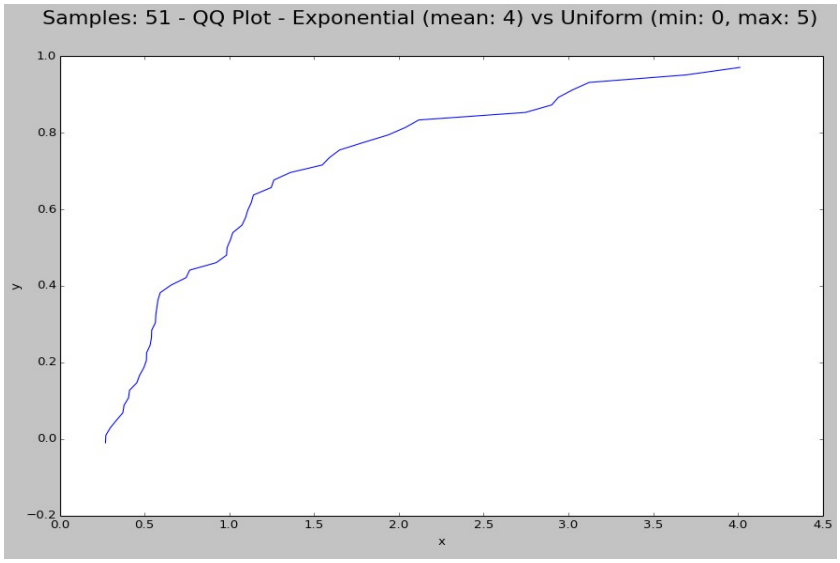


Grafico quantile-quantile distribuzione esponenziale – distribuzione uniforme





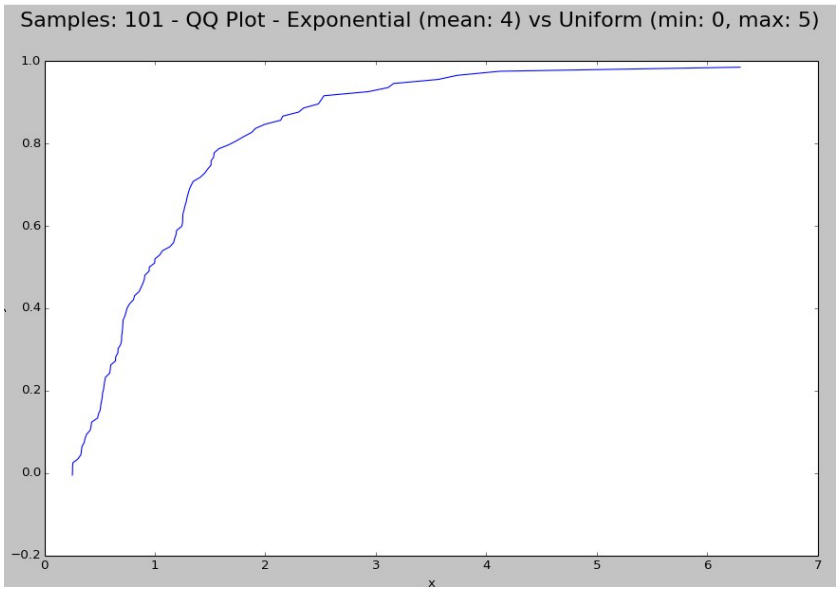
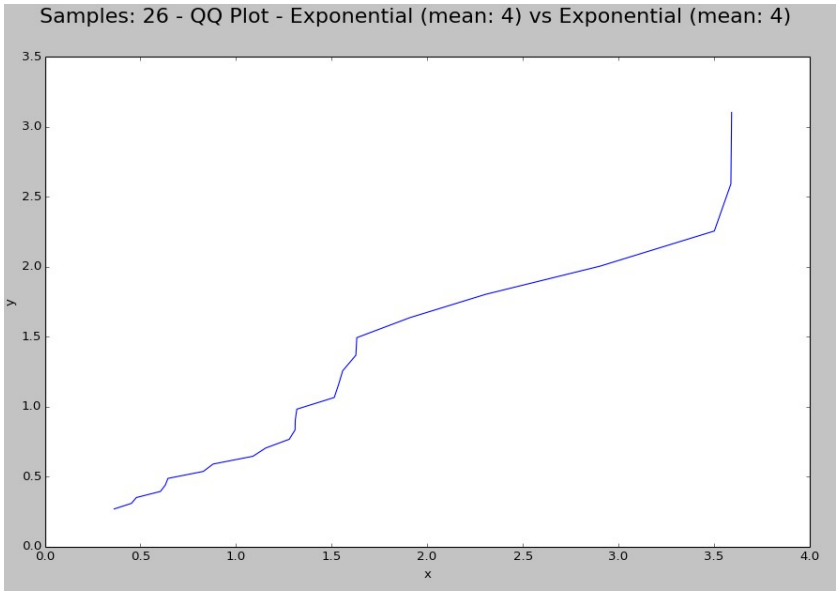
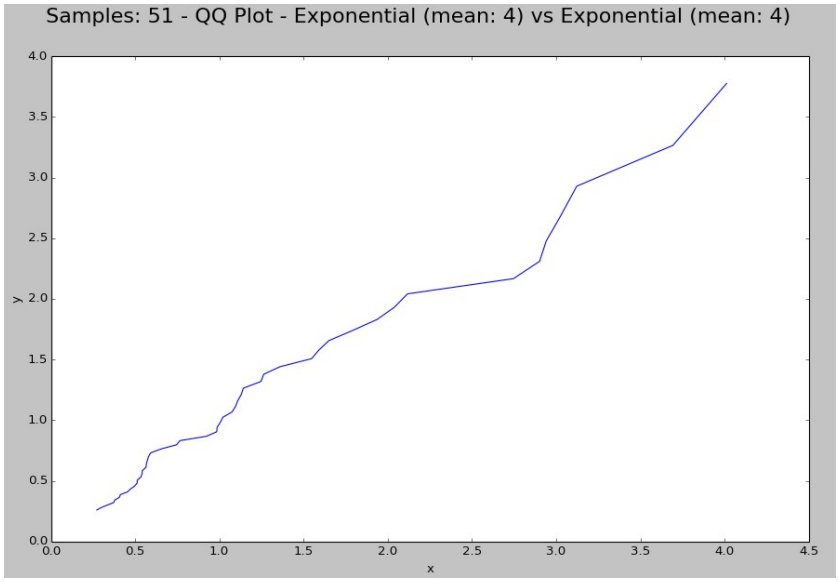


Grafico quantile-quantile distribuzione esponenziale - distribuzione esponenziale





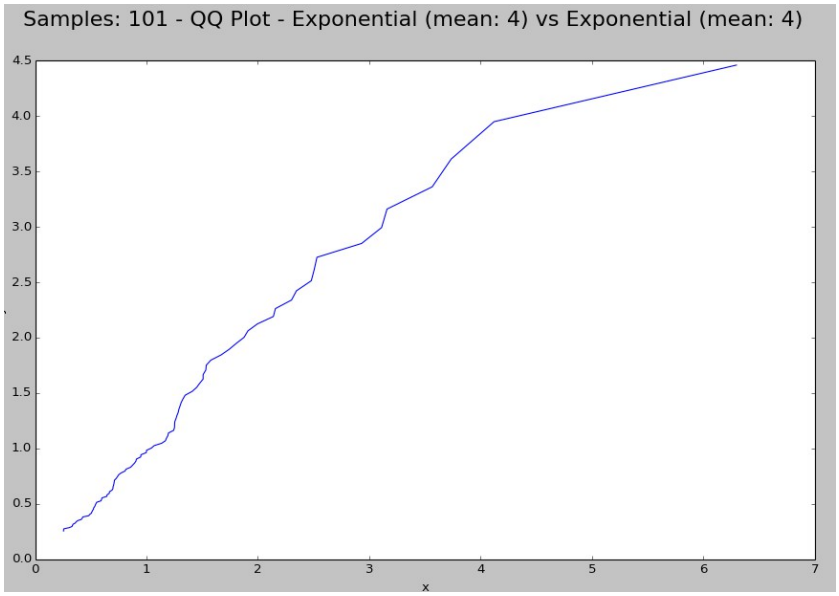
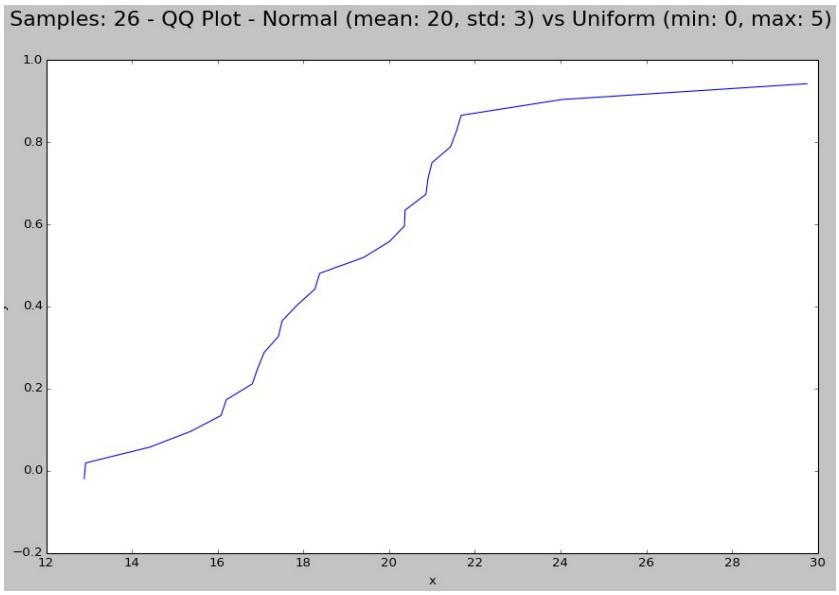
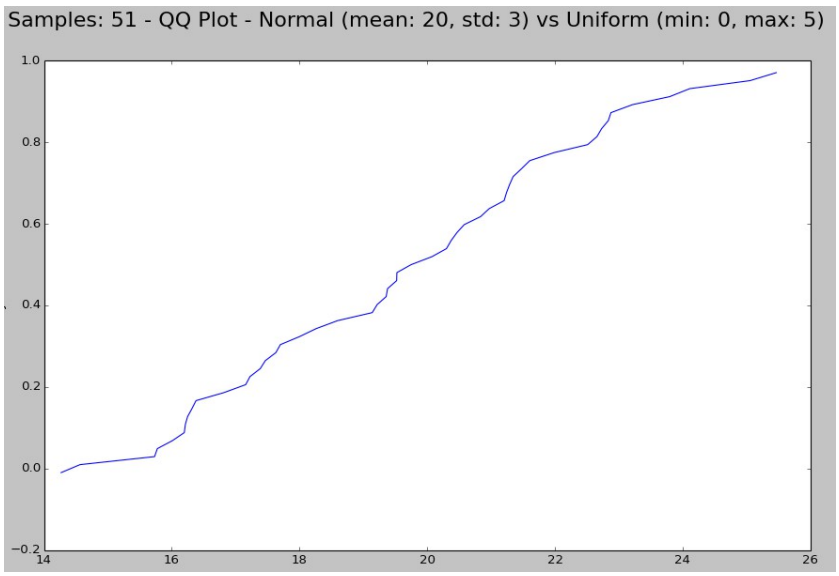


Grafico quantile-quantile distribuzione normale – distribuzione uniforme





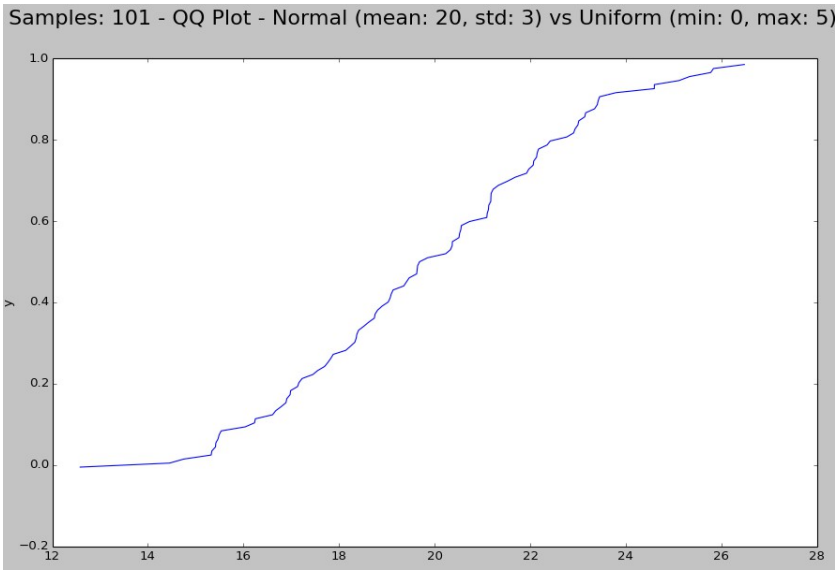
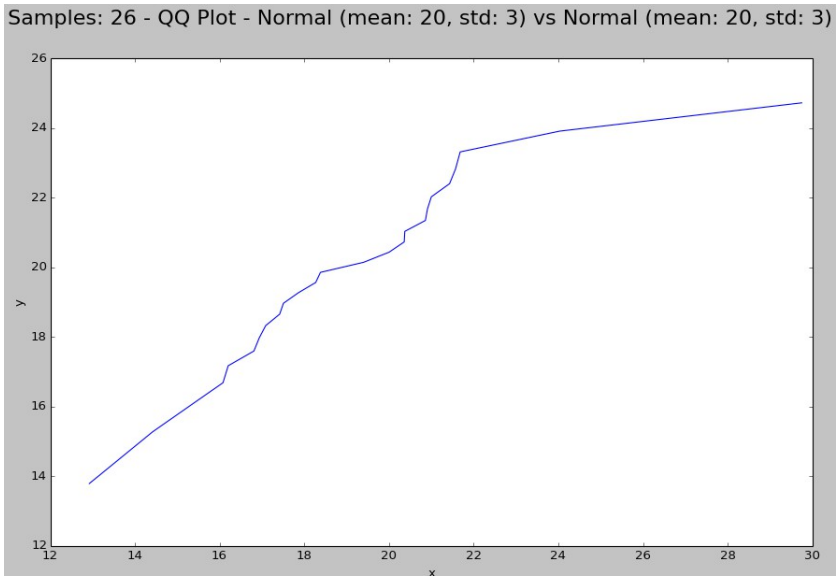
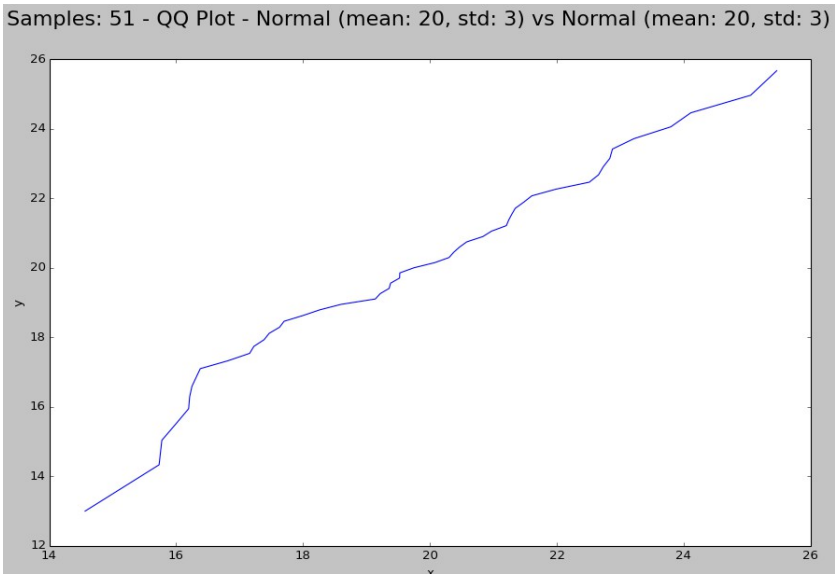
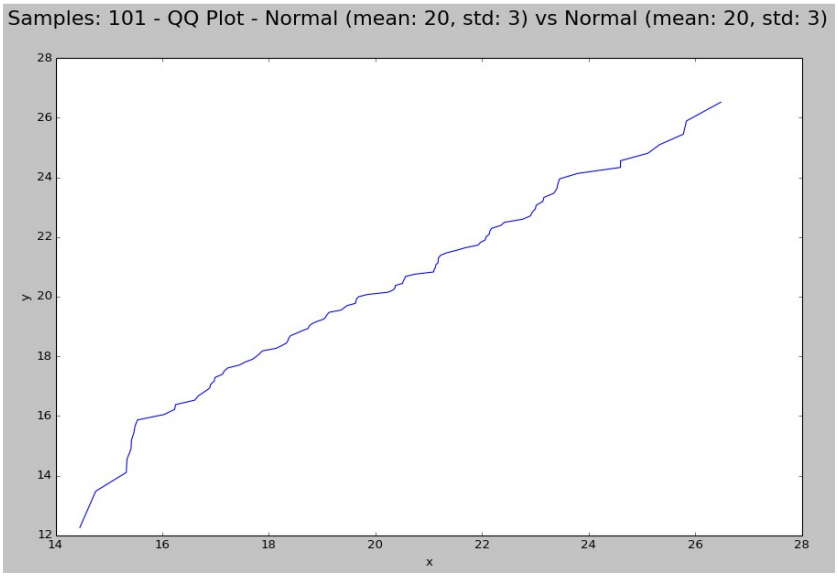


Grafico quantile-quantile distribuzione normale – distribuzione normale







Test del chi-quadro (alpha: 0.1)

# campioni	1° distribuzione	2° distribuzione	# rifiuti
26	Exponential (mean: 4)	Uniform (min: 0, max: 5)	48/100
51	Exponential (mean: 4)	Uniform (min: 0, max: 5)	54/100
101	Exponential (mean: 4)	Uniform (min: 0, max: 5)	54/100
26	Exponential (mean: 4)	Exponential (mean: 4)	1/100
51	Exponential (mean: 4)	Exponential (mean: 4)	0/100
101	Exponential (mean: 4)	Exponential (mean: 4)	0/100
26	Normal (mean: 20, std: 3)	Uniform (min: 0, max: 5)	100/100
51	Normal (mean: 20, std: 3)	Uniform (min: 0, max: 5)	100/100
101	Normal (mean: 20, std: 3)	Uniform (min: 0, max: 5)	100/100
26	Normal (mean: 20, std: 3)	Normal (mean: 20, std: 3)	0/100
51	Normal (mean: 20, std: 3)	Normal (mean: 20, std: 3)	0/100
101	Normal (mean: 20, std: 3)	Normal (mean: 20, std: 3)	0/100

Test di Kolomogorov-Smirnov (alpha: 0.1)

# campioni	1° distribuzione	2° distribuzione	# rifiuti
26	Exponential (mean: 4)	Uniform (min: 0, max: 5)	100/100
51	Exponential (mean: 4)	Uniform (min: 0, max: 5)	100/100
101	Exponential (mean: 4)	Uniform (min: 0, max: 5)	100/100
26	Exponential (mean: 4)	Exponential (mean: 4)	5/100
51	Exponential (mean: 4)	Exponential (mean: 4)	3/100
101	Exponential (mean: 4)	Exponential (mean: 4)	2/100
26	Normal (mean: 20, std: 3)	Uniform (min: 0, max: 5)	100/100
51	Normal (mean: 20, std: 3)	Uniform (min: 0, max: 5)	100/100
101	Normal (mean: 20, std: 3)	Uniform (min: 0, max: 5)	100/100
26	Normal (mean: 20, std: 3)	Normal (mean: 20, std: 3)	11/100
51	Normal (mean: 20, std: 3)	Normal (mean: 20, std: 3)	11/100
101	Normal (mean: 20, std: 3)	Normal (mean: 20, std: 3)	9/100

Implementazione del simulatore

Il simulatore è stato scritto in Python (versione 3.4 dell'interprete) con l'ausilio dell'*IDE* gratuito *PyCharm* (*Community Edition*). Per realizzarlo sono state utilizzate alcune librerie esterne gratuite, in modo particolare *SimPy*, libreria per la simulazione ad eventi discreti, *SciPy* e *NumPy*, librerie per il calcolo scientifico (utili per le distribuzioni di probabilità, test statistici, ecc.) e *Matplotlib*, libreria per la realizzazione di grafici di varia natura. Tutte le librerie sono state impiegate nelle loro versioni compatibili con l'interprete Python.

Di seguito sarà illustrato il codice relativo all'attivazione degli eventi nei processi, dal momento che è di maggiore interesse per il caso di studio; per maggiori dettagli su come è stata effettuata l'implementazione del simulatore si rimanda alla annessa documentazione del codice, generata con il *tool Doxygen* (versione 1.8.7).

Tenendo a mente che, nel sistema, un lavoro viene dapprima generato, poi accodato nella stazione di competenza, infine svolto da un lavoratore ed eventualmente accodato nella stazione successiva, risulta agevole illustrare come vengono codificate tali classi nell'ordine inverso a quello cronologico.

In breve, ciascun lavoratore attende periodicamente l'evento *start_working* che gli comunica quando svolgere un lavoro; ciascuna stazione, invece, attende periodicamente l'evento *job_arrived* che gli comunica quando attivare un suo lavoratore inattivo.

Lo sviluppo dell'interazione è il seguente:

il *JobsGenerator* genera un lavoro →

il generatore innesca l'evento *job_arrived* sulla stazione di competenza se ha lavoratori disponibili →

la stazione innesca l'evento *start_working* sul lavoratore disponibile

Ogni lavoratore è modellato da un'istanza della classe *Worker.* Ciascun lavoratore possiede, tra le altre cose, un attributo che rappresenta l'evento da attendere. Il codice esemplificato è il seguente:

```
class Worker:

    def __init__(self, shop, env, station, id):
        …
        self.start_working_event = self._env.event()

    def run(self):
        while True:
            yield self.start_working_event
            job = self.start_working_event.value
            self.start_working_event = self._env.event()
```

```
…
yield self._env.timeout(service_time)
next_station=self._shop.get_next_station(self._station.get_id(),
                                                    job.type)
if next_station is not None:
      next_station.add_job(job)
      if next_station.has_available_worker():
          next_station.job_arrived_event.succeed()
…
```

In modo ciclico, il lavoratore attende l'evento che lo farà cominciare a lavorare; preleva da esso il lavoro da compiere; resetta l'evento, in modo da poterne ricevere altri in futuro; svolge il lavoro (attendendo una quantità di tempo pari al tempo di servizio); controlla se il percorso del lavoro corrente prevede una nuova stazione e, in tal caso, accoda il lavoro alla coda della stazione successiva e innesca l'evento che notifica la stazione del suo arrivo.

Similmente, ogni stazione è modellata da un'istanza della classe *Station.* Ciascuna stazione possiede, tra le altre cose, un attributo che rappresenta l'evento da attendere. Il codice esemplificato è il seguente:

```
class Station:
     def __init__(self, shop, env, id, distrib):
          …
          self.job_arrived_event = self._env.event()
          self._idle_workers_list = []

     def run(self):
          while True:
               yield self.job_arrived_event
               self.job_arrived_event = self._env.event()
               curr_worker = self._idle_workers_list[0]
               curr_worker.start_working_event.succeed(self._job_queue.get())
               …
```

In modo ciclico, la stazione attende l'evento che le notifica l'arrivo di un lavoro; resetta l'evento, in modo da poterne ricevere altri in futuro; seleziona il primo lavoratore disponibile dalla lista di quelli inattivi e su di esso innesca l'evento che lo farà cominciare a lavorare, inserendo all'interno dell'evento il lavoro da svolgere (il primo nella coda dei lavori in attesa di essere svolti).

Il processo che genera nuovi lavori è codificato nella classe *JobsGenerator*. Durante tutta la simulazione tale processo si limita a ripetere una sequenza di azioni, esemplificata da questo frammento di codice:

```
while True:
     new_job = Job(JobsGenerator.counter, JobsGenerator._get_job_type(),
                                                                  self._env.now)
     first_station = self._shop.get_station(Constants.FIRST_STATIONS[new_type])
     first_station.add_job(new_job)
     if first_station.has_available_worker():
          first_station.job_arrived_event.succeed()
     yield self._env.timeout(JobsGenerator._inter_time())
```

Come si può intuire, il generatore crea un nuovo lavoro (e gli assegna un opportuno tipo); individua qual è la prima stazione che dovrà servirlo e gli accoda il lavoro; controlla se la stazione ha qualche lavoratore libero e, in caso affermativo, innesca l'evento per notificare l'arrivo di un lavoro; infine attende per un certo periodo di tempo l'arrivo di un nuovo lavoro.

Verifica e validazione del simulatore

Al termine della fase di realizzazione del sistema, sono stati effettuati una serie di test allo scopo di verificare la presenza di eventuali falle o malfunzionamenti. Di seguito vengono riportati i principali test effettuati e, successivamente, vengono illustrate una serie di situazioni d'errore che hanno comportato la modifica di alcune parti del sistema, al fine di rispettare le richieste del caso di studio.

Test dei semi

Per verificare che l'output della simulazione non fosse dipendente dalla particolare sequenza di numeri pseudo-casuali generati, sono state condotte diverse simulazioni ciascuna con un seme diverso. Tutti gli intervalli di confidenza prodotti sono risultati essere molto simili all'output ottenuto utilizzando il seme di default (12345), i cui risultati sono presentati più avanti. Pertanto è possibile affermare che il seme della distribuzione non influenza l'output della simulazione.

Test degenerativo su tempi lunghi

Effettuando la simulazione per tempi molto lunghi (tempo di start-up: *200*, tempo totale di simulazione: *8000*) vengono generati e svolti circa *6300* lavori. Si nota immediatamente che aumenta sia il tasso di utilizzo dei lavoratori nelle stazioni, sia il tempo medio di permanenza dei lavori nel sistema. Questo risultato si può spiegare ipotizzando che il maggior numero di lavori generati abbia causato un allungamento delle code alle stazioni: i lavoratori passeranno meno tempo inattivi (perché le code non saranno quasi mai vuote) e i lavori, come conseguenza, dovranno attendere di più nelle code. Riportiamo di seguito un confronto tra gli intervalli di confidenza (prima nella situazione a regime, poi nella situazione limite):

Entità misurata	Situazione a regime	Situazione limite
Utilizzo lavoratori – Stazione 1	0.6600 <= p <= 0.6830	0.870813 <= p <= 0.8781
Utilizzo lavoratori – Stazione 2	0.5653 <= p <= 0.6394	0.7638 <= p <= 0.7892
Utilizzo lavoratori – Stazione 3	0.6708 <= p <= 0.7025	0.87331 <= p <= 0.8885
Utilizzo lavoratori – Stazione 4	0.4583 <= p <= 0.4991	0.6162 <= p <= 0.6309
Permanenza lavori – Tipo 1	203.8745 <= p <= 210.6824	245.7618 <= p <= 249.3182
Permanenza lavori – Tipo 2	128.6816 <= p <= 133.8605	159.6756 <= p <= 161.3287
Permanenza lavori – Tipo 3	120.0250 <= p <= 125.6881	153.7697 <= p <= 156.6609
Permanenza lavori – Tipo 4	71.0874 <= p <= 74.7698	89.94752 <= p <= 91.7520

Test degenerativo su tempi brevi

Effettuando la simulazione per tempi molto brevi (tempo di start-up: *0*, tempo totale di simulazione: *100*) si è riscontrato che non viene raccolto un numero di dati sufficientemente grande per poter calcolare gli intervalli di confidenza. Non a caso, dunque, il caso di studio richiede un tempo di simulazione maggiore. In ogni caso, nel breve tempo della simulazione, non si sono riscontrate anomalie nell'esecuzione del simulatore.

Caso di debug: mancato assegnamento di un lavoro alla stazione successiva

Uno dei problemi più significativi fra quelli riscontrati era costituito dalla mancata presenza di un meccanismo che permettesse ad ogni lavoratore di assegnare il lavoro appena completato alla stazione successiva che avrebbe dovuto prenderlo in carico.

Viene riportato di seguito il codice della classe *Worker* prima e dopo la correzione:

```
…
service_time = self._station.get_service_time(job.type)
yield self._env.timeout(service_time)
end_time = self._env.now
next_station=self._shop.get_next_station(self._station.get_id(), job.type)
next_station.add_job(job)
```

Come è ben visibile, il passaggio del lavoro corrente alla stazione successiva veniva fatto senza effettuare alcun tipo di controllo sull'effettiva esistenza della stazione, per tale ragione, si poteva incorrere in anomalie dovute all'assenza di un riferimento ad un oggetto valido di tipo *Station* che potesse soddisfare la richiesta. Di seguito viene presentata la parte di codice (viene trascurata la parte non significativa ai fini del caso di debug in esame) che provvede a porre rimedio alla situazione d'errore che si era generata:

```
…
next_station= self._shop.get_next_station(self._station.get_id(), job.type)
if next_station is None:
     job.end_t = end_time
     JobsGenerator.served_job_list.append(job)
else:
     next_station.add_job(job)
     if next_station.has_available_worker():
          next_station.job_arrived_event.succeed()
self._station.add_idle_worker(self)
```

Non introducendo tale controllo, i lavori, dopo esser stati completati dalla prima stazione di competenza, non avevano più modo di terminare il loro percorso perché non transitavano da tutte le stazioni. Risultavano, inoltre, inevitabilmente non completati, alterando le statistiche complessive calcolate.

Caso di debug: tempi di fine servizio settati in modo errato

Un'ulteriore anomalia riscontrata riguardava l'assegnazione del tempo di fine servizio per ogni specifico lavoro. Nel momento in cui si procede ad avvalorare il tempo di fine servizio di un determinato lavoro, non è semplicemente necessario valutare quale sia il tempo di inizio e il tempo totale impiegato dal lavoratore per poter portare a termine quel determinato compito, bensì è necessario dapprima valutare se la stazione nella quale il lavoratore si trova, risulta essere l'ultima stazione del suo percorso.

Infatti, per poter ovviare al problema è stato introdotto un controllo che prevenga tale anomalia, il cui codice viene brevemente riportato di seguito (metodo run() della classe *Worker*):

```
job.wait_t += self._env.now - job.start_t
service_time = self._station.get_service_time(job.type)
yield self._env.timeout(service_time)
end_time = self._env.now
if self._env.now > Constants.STARTUP_TIME:
     self.total_work_time += service_time
next_station=self._shop.get_next_station(self._station.get_id(), job.type)
if next_station is None:
     job.end_t = end_time
     if job.start_t > Constants.STARTUP_TIME:
          JobsGenerator.served_job_list.append(job)
```

Come è possibile vedere, dopo aver memorizzato il tempo di attesa totale del lavoro all'interno del sistema e del tempo di servizio, viene salvato solo momentaneamente il tempo di fine servizio, che sarà settato solo dopo aver verificato che il lavoro abbia completato il suo percorso.

Caso di debug: generazione dei tempi di servizio

Un altro errore riscontrato durante la realizzazione del sistema e del quale si è subita capita la natura, è stato quello legato alla generazione dei tempi di servizio dei vari lavori nel sistema.

Questi ultimi venivano prodotti secondo una distribuzione normale che, essendo una funzione simmetrica, ritornava dei valori che potevano risultare anche negativi, comportando quindi degli errori in fase di esecuzione del processo di simulazione.

Ritornando il valore assoluto del tempo di servizio tale problema è stato risolto, preservando la correttezza logica dell'intero processo.

Caso di debug: elementi insufficienti per il calcolo delle statistiche in output

Un'ulteriore situazione di errore riscontrata nel corso degli esperimenti di verifica del sistema è stata quella relativa alla mancata elaborazione delle statistiche risultanti dall'intero processo di simulazione. Per poter calcolare delle medie si ha la necessità di avere a disposizione un numero sufficiente di campioni. Nel caso in cui questo non fosse vero, si andrebbe incontro a problematiche quali divisioni per zero o altre situazioni anomale in fase di calcolo.

Per tale ragione, nel momento in cui si procedeva nel calcolare le statistiche sui dati di output del

sistema dopo aver effettuato una sola esecuzione (*run)*, il programma produceva dei valori indesiderati.

Di seguito viene mostrata la parte di codice che ha il compito di assicurarsi che il numero di esecuzioni sia sufficiente per calcolare le statistiche:

```
def run(stats, exercise_num):

…
if Constants.NUMBER_OF_RUNS > 1:

…
else:
print('No stats available: an insufficient number of run has been
                                                                executed.')
```

Analisi statistica dell'output

Come richiesto dal caso di studio, la simulazione è stata ripetuta un certo numero di volte per elaborare delle statistiche. Tutti gli intervalli di confidenza da calcolare, già elencati nel paragrafo introduttivo, sono qui di seguito riportati (tra parentesi figura la confidenza dell'intervallo).

Esercizio 13: utilizzo medio dei lavoratori per ogni stazione (97,5%)

Stazione 1: 0.6600997320434862 <= p <= 0.6830659222994916
Stazione 2: 0.5653318293949807 <= p <= 0.6394524702749169
Stazione 3: 0.6708834143712209 <= p <= 0.7025563472178579
Stazione 4: 0.4583639080972669 <= p <= 0.49913724828041195

Esercizio 13: tempo medio di permanenza nel sistema di un tipo di lavoro (95%)

Tipo 1: 203.87455854788146 <= p <= 210.68247253450977
Tipo 2: 128.6816682939167 <= p <= 133.8605503692567
Tipo 3: 120.02501233379773 <= p <= 125.68818426236477
Tipo 4: 71.087477922784 <= p <= 74.76988017115487

Esercizio 14: tempo medio di permanenza nel sistema di un tipo di lavoro – ordinamento per tipo (97,5%)

Tipo 1: 71.087477922784 <= p <= 74.76988017115487
Tipo 2: 0 <= p <= 0
Tipo 3: 0 <= p <= 0
Tipo 4: 0 <= p <= 0

Esercizio 14: tempo medio di permanenza nel sistema di un tipo di lavoro – ordinamento FIFO (97,5%)

Tipo 1: 202.99310343411372 <= p <= 211.5639276482775
Tipo 2: 128.0111321946248 <= p <= 134.5310864685486
Tipo 3: 119.29177277372308 <= p <= 126.42142382243942
Tipo 4: 70.61069864808228 <= p <= 75.2466594458566

Identificazione della soluzione e presentazione dei risultati

Il caso di studio richiedeva di effettuare la simulazione del sistema provando due politiche di gestione delle code di lavori nelle stazioni: la politica FIFO e quella per priorità rispetto al tipo di lavoro (a tipo inferiore corrispondeva maggiore priorità).

Osservando gli intervalli di confidenza precedentemente mostrati, una cosa è immediatamente balzata all'attenzione: con la politica di ordinamento per priorità, il sistema sembra svolgere unicamente lavori di tipo 1 (intervalli come *0 <= p <= 0* sono il risultato dell'assenza di dati).

Inizialmente si è pensato ad un errore nell'implementazione del simulatore, ma, guardando il trace che era stato generato, si è potuto evincere che, al contrario, lavori di ogni tipo venivano correttamente generati e, in una fase iniziale, anche svolti.

Analizzando più a fondo la situazione è stato possibile individuare il problema.

I lavori di tipo 1 godono di maggiore priorità e, in aggiunta, sono generati in quantità superiori agli altri tipi (0.4 vs 0.3 vs 0.2 vs 0.1).

Nella fase iniziale della simulazione, nonostante vengano creati parecchi lavori di tipo 1 (che oltrepassano i lavori degli altri tipi), il maggiore numero di lavoratori inattivi permette di servire contemporaneamente tutti questi lavori più una piccola quantità di lavori di tipo 2, 3 o 4.

Col passare del tempo, tuttavia, i lavori di tipo 1 continuano ad aumentare in numero, fino a non trovare più lavoratori inattivi che possano svolgerli: in questo momento cominciano ad essere messi in coda. Insieme a loro finiscono tutti gli altri lavori che arrivano nel sistema, obbligati a dare priorità a quelli di tipo 1. La situazione che pertanto si verifica è una sorta di “tappo”, poiché i lavori di tipo 1 continuano ad arrivare e ad oltrepassare tutti gli altri in coda, che non saranno di fatto mai serviti. Il sistema troverà infatti, in testa alla coda, sempre lavori di tipo 1.

Impostando un tempo di start-up sufficientemente grande (come quello del caso di studio) si creano i presupposti per il verificarsi del fenomeno appena illustrato. Se, al contrario, si imposta un tempo di inizializzazione molto piccolo, allora si riuscirà a raccogliere le statistiche sui pochi lavori di tipo 2, 3 e 4 che sono stati svolti dal sistema prima che quelli di tipo 1 “prendessero il sopravvento”: in tal caso gli intervalli di confidenza non saranno tutti del tipo *0 <= p <= 0*, ma ci saranno, seppur poche, informazioni sufficienti per il calcolo di alcune statistiche. Di seguito viene presentata una prova (tempo di start-up pari 0):

Tipo 1:	184.5271356217822 <= p <= 208.51192989393698
Tipo 2:	86.61274400265009 <= p <= 111.04043348911969
Tipo 3:	72.4373310460917 <= p <= 90.32916104426671
Tipo 4:	0 <= p <= 46.61918979737055

Alla luce di quanto detto, la politica di gestione FIFO risulta l'unica ragionevolmente accettabile.

Tutti gli altri intervalli di confidenza (occupazione dei lavoratori e tempo di medio di permanenza nel sistema dei lavori) producono risultati coerenti con i presupposti della simulazione: le stazioni visitate più spesso hanno un maggiore tasso di utilizzo ed i lavori con percorso più lungo restano più tempo nel sistema.