Chiara Bruschi
Davide Ferrari
Federico Motta
Filippo Muzzini
Davide Paganelli
Davide Sapienza

```
*****  
LINIALIZZAZIONE DI AESOP-ACP  
*****
```

Descrizione initialize():

Il metodo initialize della classe Simulator legge e interpreta i file schedule.xlsx e agent_graphs.dot per generare gli agenti e gli eventi in essi descritti. initialize() invoca tutti gli altri metodi di inizializzazione (descritti in seguito) per la creazione della simulazione e svolge tutte le operazioni per farla eseguire.

Salva lo stato del simulatore nel modulo commonstate. In particolare vengono settate le variabili:

- cmstate.r: generatore di numeri random;
- cmstate.verbose: il livello di dettaglio dei messaggi di log;
- cmstate.until: numero massimo di passi che il simulatore può effettuare;
- cmstate.start_size: dimensione iniziale del gruppo di agenti.

Inoltre viene salvato, sempre in cmstate, un riferimento all'istanza corrente di Simulator (cmstate.sim).

Vengono presentate in dettaglio le operazioni che il metodo svolge:

- readGraph(): legge il file agent_graphs.dot e ritorna il grafo corrispondente in agentsGraph, un oggetto di tipo networkx.DiGraph (Grafo diretto)

La funzione “readGraph()” prende in input il file .dot che descrive il grafo in formato graphviz e ne ritorna una struttura dati attraverso i moduli pygraphviz (per l'interpretazione del file .dot) e networkx (per la manipolazione di grafi). Tale struttura dati è la rappresentazione di un grafo indiretto ai cui nodi sono state opportunamente assegnate le proprietà indicate nel file .dot (le stringhe rappresentanti numeri vengono convertite in float, mentre le restanti proprietà restano di tipo string).

Successivamente viene salvata la variabile globale “until” (“cmstate.until”) che può assumere come valore il massimo intero rappresentabile in aritmetica di macchina (maxint), se non viene inizializzata da linea di comando o come proprietà del grafo nel file .dot. Essa rappresenta il numero di iterazioni da compiere nella simulazione.

- `agentFactory()`: chiamata solo se `agentsGraph` è definito. Prende i nodi di questo grafo e per ogni nodo crea un agente di tipo `cls_type` con gli attributi definiti in `agent_graphs.dot`.
 - `factory()`: tramite il nome della classe verifica che il relativo oggetto sia presente in `__class_cache` (un dizionario che memorizza gli oggetti classe per aumentare le performances) e lo ritorna al chiamante. Nel caso l'oggetto non sia presente in `__class_cache`, viene inserito.

Se la generazione del grafo va a buon fine, si procede con l'istanziamento degli agenti attraverso la funzione "agentFactory".

In quest'ultima avviene il riempimento della struttura dati

"`cmstate.sim.agents`" al fine di memorizzare l'oggetto rappresentante l'agente.

Da notare come "`cmstate.sim`" rappresenti l'istanza della classe "Simulator".

Per inizializzare tale oggetto si utilizza la funzione "factory", la quale implementa un meccanismo di caching per trovare più velocemente le classi.

Tale procedimento è assimilabile al design pattern chiamato "Factory", ovvero l'applicazione ricava dinamicamente la classe da istanziare, analizzando l'attributo "classType" di ogni nodo presente nel file ".dot".

Successivamente avviene la rimozione di eventuali agenti in background da "`cmstate.sim`".

Una volta completata la creazione e l'inizializzazione del grafo dal file .dot, la funzione "agentFactory()" procede con la creazione degli agenti, che corrispondono ai nodi del grafo generato. Questi sono contenuti in un dictionary dove le keys sono i nomi degli agenti e i values sono le rispettive caratteristiche.

Le caratteristiche principali sono "classType" (default="Agent"), "capacity" (quantità di ordini che l'agente è in grado di processare) e "tags" (le capacità di un agente di svolgere determinati compiti). Per ogni nodo agente, il nome ("name") e i "tags" vengono inseriti in un altro dictionary usato come parametro di input per la funzione "factory".

La funzione "factory()" si occupa della creazione di un'istanza della classe dell'agente, opportunamente inizializzata con le caratteristiche sopra indicate. In output fornisce l'istanza, la quale viene inserita all'interno del dictionary degli agenti accessibile globalmente tramite "`cmstate.sim.agents`". Ogni agente del dictionary è accessibile tramite il nome dell'agente stesso (es: "`cmstate.sim.agents['nomeagente']`").

- `initializeRouteGraph()`: chiamata solo se `agentsGraph` è definito. Crea l'oggetto `routeGraph`, un grafo indiretto costruito a partire da `agentsGraph`. Aggiunge inoltre l'attributo `load` ad ogni nodo di `routeGraph`. L'attributo `load` contiene il numero di elementi presenti nella coda in ingresso del nodo stesso.

Il metodo "`initializeRouteGraph()`" copia il grafo degli agenti ("`agentsGraph`") per creare un grafo delle route ("`routeGraph`", `route`=percorso tra i nodi). Ai nodi del grafo viene anche aggiunta la proprietà "load" corrispondente alla lunghezza delle code in entrata di

ogni agente (“incomingQ”), dove sono inseriti gli ordini in attesa di essere eseguiti.

- selectMgr: attributo di Simulator che contiene la policy per la scelta dell'agente a cui dare il prossimo lavoro. Di default è ShortestQueueAgentManagers().

Il settaggio della policy di scheduling dei task agli agenti viene realizzato attraverso il caricamento di una classe rappresentante l'opportuna policy da attuare, anche in questo caso la scelta di tale classe risulta dinamica e può quindi essere indicata dall'utente. Di default avviene il caricamento della classe "ShortestQueueAgentManager", la quale permette di schedulare il lavoro all'agente adibito a tale compito avente la coda più scarica.

La classe dello schedule manager deve essere indicata al momento dell'esecuzione del programma da terminale. Questa classe si occupa principalmente di scegliere l'agente cui schedulare gli ordini da soddisfare.

La scelta viene fatta al momento della chiamata del metodo “doOrder()” (propria di un agente), gestita da un “orderDistiller” (coinvolto al momento della manipolazione del file .dot). Viene data precedenza all'agente con coda di attesa dei task meno piena e, in particolare, il controllo sulla coda “incomingQ” viene fatto dalla funzione “agentsNextTasks()”, all'interno del file parsers.py.

- initializeScheduleManager(): legge, interpreta e rende disponibili i dati all'interno del file schedule.xlsx tramite il metodo readXLSSchedule() della classe EnterpriseScheduleManager.

La funzione "readXLSSchedule" presente all'interno della funzione "initializeScheduleManager" effettua il parsing del file "schedule_sec.xlsx" e crea due dizionari: "scheduleTable" e "macroTable".

Il primo contiene l'indicazione delle macro create con il rispettivo campo rappresentante ogni quanto avviene la relativa esecuzione (in questo caso si ha la sola macro "production"), mentre il secondo mantiene le informazioni relative agli ordini specifici di ogni macro. La funzione riempie dapprima i due dizionari analizzando il foglio "schedule" del file excel, successivamente apre il foglio relativo ad ogni macro e popola la relativa entry della "macroTable" con i specifici ordini.

Sullo schedule manager viene invocato il metodo “readXLSSchedule()” il cui compito è aprire tramite la libreria “openpyxl” il file Excel (di default impostato a “schedule.xlsx”, ma più utile se impostato manualmente da linea di comando specificando il file disponibile più aggiornato, ad esempio “schedule_sec.xlsx” per l'esempio “embroidery”). Di questo file viene cercato il foglio “schedule” che indica quali ordini eseguire e la loro successione. Un ordine può anche essere definito da una macro, indicata in un altro foglio all'interno del file Excel, e la quale può indicare con che periodo l'azione va eseguita (“every”, in timestamps) oppure una

condizione che determina quando fermarne l'esecuzione ("until").

Nell'invocazione del metodo "initializeScheduleManager()" viene caricata dinamicamente in memoria (tramite la funzione "get_class()") la classe che implementa lo schedule manager (che di default è "aesop.schedule.ScheduleManager" e viene implementata nel modulo "schedule.py"). Tale classe viene poi utilizzata per creare un'istanza di schedule manager ("scheduleMgr") che può essere un oggetto di tipo "ScheduleManager" o una sua classe derivata (come "EnterpriseScheduleManager", anch'essa definita nel modulo "schedule.py")

In seguito viene eseguito il parsing di altri eventuali fogli Excel: nel caso essi abbiano come nome "groups" o "simcontrol" vengono usati metodi dedicati ("parseGroupSheet()" e "parseSimControlSheet()"), mentre per tutti gli altri tipi di foglio vi è un ciclo che fa il parsing in sequenza delle linee e le inserisce nel dictionary "macroTable" (in cui le keys sono i nomi dei fogli ed i values i loro contenuti).

- scheduleMgr.parseInitSheet(): se esiste, interpreta il foglio di calcolo "init" del file schedule.xlsx e crea una rete a partire dalle informazioni presenti sul foglio "init" (tipo di agenti, topologia della rete, proprietà degli agenti, eventi da generare per gli agenti).

Tale foglio può contenere diverse istruzioni relative alla creazione del grafo degli agenti, eventualmente necessario per la generazione automatica di molti agenti. All'interno di suddetto metodo è possibile settare la classe degli agenti, inizializzare la rete tramite un metodo arbitrario della libreria networkx, settare le proprietà ai diversi agenti attraverso l'invocazione di metodi sull'agente stesso o tramite il settaggio di un numero nel campo "what" dell'agente. Inoltre è possibile settare il random seed dell'applicazione (cmstate.r.seed). Nel caso alcune righe non contenessero istruzioni relative a quelle descritte in precedenza, il contenuto della riga stessa sarà elaborato dal metodo "parseXLSLine" e successivamente passato a "generateEvents", ovvero verrà trattata come la descrizione di un ordine impartito ad un agente (o gruppo di agenti).

Da notare che le operazioni descritte in precedenza avvengono dopo l'eventuale creazione di un grafo descritto nel file ".dot" e, per come è fatto il codice, tale grafo viene sovrascritto da questa funzione, quindi se vi è un foglio "init" del file xlsx contenente le istruzioni per la generazione del grafo, un eventuale file ".dot" verrà dapprima letto ma successivamente l'applicazione non ne terrà conto poichè tale grafo verrà sovrascritto da quello creato in questa funzione.

Determinati gli agenti, è possibile inizializzare la variabile globale "start_size" ("cmstate.start_size") che indica la dimensione iniziale della rete (posta pari al numero di agenti).

- scheduleMgr.generateScheduleEvents(): genera e schedula, a partire dai fogli

“schedule” e “production” di schedule.xlsx, gli eventi per gli agenti. generateScheduleEvents scorre ogni riga del foglio “schedule” e differenzia il suo comportamento tra righe che iniziano per “macro” e le restanti.

- generateEvents(): per ogni riga, genera un evento per ogni agente. Utilizza le parole chiave every e until per comunicare allo scheduler come gestire gli eventi.
- ScheduleEvent(): classe che istanzia un oggetto schedulabile.

La funzione "generateScheduleEvents" (relativa allo schedule manager), ad alto livello rappresenta un meccanismo di associazione di eventi agli agenti al fine di impedire loro di rimanere in assenza di lavoro.

Per realizzare quanto detto, in primis si costruiscono opportune strutture dati contenenti le informazioni sia relative all'intervallo di tempo in cui invocare una macro (parametro "every"), che inerenti alla lista degli ordini specificati nella macro stessa (ovvero avviene una manipolazione dei dati raccolti nei dizionari "scheduleTable" e "macroTable" costruiti nella sezione precedente), e poi si procede alla generazione degli eventi stessi.

Il simulatore genera nuovi eventi ("self.scheduleMgr.generateScheduleEvents()") per ogni entry presente nel file di scheduling degli ordini richiesti fornito al momento dell'esecuzione da terminale. Il concetto di “evento” si riassume nell'azione di risveglio di un agente e successiva assegnazione dell'ordine da eseguire. Il metodo “generateEvents()” da cui parte la creazione degli eventi si trova all'interno del file “schedule.py” e a sua volta genera una istanza della classe “ScheduleEvent” (definita nel file “acp.py”) per ogni evento assegnato ad un agente. Tramite il metodo “locateAgents()” si ottiene il numero di agenti che devono eseguire la richiesta corrente (da uno fino ad un gruppo di agenti).

Dall'istanza della classe “ScheduleEvent” creata poi, per ogni ciclo su un eventuale campo “every” indicato nel file Excel di input, vengono innestate ulteriori istanze della classe, figlie di quella principale (il limite è comunque imposto dalla variabile globale “until” oppure di default l'intero più grande rappresentabile dall'aritmetica di macchina). Per controllare se sia necessario creare istanze innestate, si utilizza il metodo “nextCallAt()”. Ogni evento creato viene gestito tramite una struttura dinamica di tipo heap con allocazioni (push) e deallocazioni (pop).

La funzione "generateEvents" non fa altro che scansionare uno alla volta tutti gli ordini della macro in esame e per ognuno di essi si effettua l'associazione di tale evento ad ogni agente in grado di eseguirlo e infine lo si aggiunge alla coda globale del simulatore.

Si noti che in tale funzione sono presenti diversi costrutti condizionali probabilmente per rispondere a molteplici condizioni in cui ci si può imbattere, ma nel nostro caso (per come è fatta la macro "production") si andrà ad eseguire solamente la porzione di codice che soddisfa la condizione di più di 4 elementi caratterizzanti un qualsiasi ordine (linea 399).

L'associazione avviene per mezzo dell'istanziamento di un oggetto relativo alla

classe "ScheduleEvent", la quale mantiene il mapping tra l'agente in questione, l'ordine ("doOrder") e i parametri della macro.

Tale classe serve di fatto per segnalare l'attivazione di un evento avvenuta ad opera dello scheduler.

Il responsabile dell'aggiunta dell'evento alla coda globale del simulatore relativa all'agente è il metodo "addEvent".

Si noti che la coda globale del simulatore è rappresentata da una nota struttura dati: una coda con priorità.

- Infine, solo nel caso in cui esista il foglio "simcontrol" nel file schedule.xlsx, svolge ulteriori operazioni. Questa parte di codice è obsoleta, nelle ultime versioni del software non viene mai eseguita.

Il codice del metodo Simulator.initialize() prosegue con un controllo che, però, non è quasi mai soddisfatto: "if scblocks != 0:" (se scblocks è diverso da zero). Dato che la variabile "scblocks" è quasi sempre 0 e dato che dopo il blocco "if" non sono presenti altre istruzioni, l'esecuzione del metodo Simulator.initialize() termina.

In particolare, ciò accade perché la lunghezza dell'attributo "controlTable" dell'oggetto "scheduleMgr" è diversa da zero solo se viene eseguito il metodo "parseSimControlSheet()", ovvero se è presente un foglio con nome "simcontrol" nel file Excel. Tuttavia ciò non accade quasi mai, dal momento che il foglio "simcontrol" è stato ereditato dalle prime versioni del programma in cui era richiesta la compatibilità con esempi (come "school") che nel tempo sono stati accantonati a causa di problemi nell'esecuzione.

Lista metodi chiamanti: initialize() viene invocata in due punti, entrambi nel "main" (startsim.py) sia nel caso in cui ci sia la GUI che in sua assenza.

Publicato con licenza Creative Commons Attribution-ShareAlike 4.0 (CC BY-SA 4.0).