UNIVERSITÁ POLITECNICA DELLE MARCHE

FACOLTÁ DI INGEGNERIA

Corso di Laurea Magistrale in
Ingegneria Informatica e dell'Automazione

Progettazione e sviluppo di un algoritmo per la compressione di file di log mediante Local Process Model

Docenti:
Dott. Potena Domenico
Dott.ssa Genga Laura

Realizzato da:
Antenucci Lucrezia
Mele Alessandro
Traini Davide

Anno Accademico 2021-2022

Indice

Capitolo 1

Introduzione

1.1 Process Mining

Il *Process Mining* è una verticalizzazione del *Data Mining* che si pone come obiettivo analizzare i processi, intesi come sequenze di attività finalizzate al raggiungimento di uno specifico obiettivo.
Esempi di processi sono l'esecuzione di un programma o le attività di un'azienda; queste ultime, anche se rigidamente sottoposte a regole o normative, spesso vengono eseguite diversamente da quanto previsto.
Se il modello di processo è troppo restrittivo, ed il personale si accorge di una metodologia migliore per svolgere le attività, allora si inizieranno a proporre delle strade alternative: se risultano efficaci ed efficienti, allora si tenderà ad adottare il nuovo modello.
Si fa *Process Mining* perché la maggior parte dei processi richiede persone che prendono delle decisioni in un contesto dove c'è alta variabilità, la quale dipende dalla metodologia di lavoro del personale.

Il*Process Mining* è basato sull'analisi dei dati di *log* provenienti dai sistemi informativi aziendali, all'interno dei quali sono contenute informazioni fortemente eterogenee; per questo motivo, sono necessarie tecniche di *Machine learning* e *Ricerca operativa* per estrarre informazioni rilevanti.
A partire dalle informazioni contenute nei sistemi informativi, è possibile ricavare un *file di log* contenente *tracce*, ovvero un insieme di eventi consecutivi che sono stati portati a termine.
In genere, un *file di log* viene rappresentato in formato *.xes* (*eXstensible Event Stream*), o *.csv*. Il modello di processo viene definito sotto forma di *rete di Petri* o *albero di processo*.
Il *Process Mining* si divide in tre tipologie di task:

- Process Discovery: si occupa di estrarre automaticamente modelli di processo; esistono diverse tecniche, come *Alpha Miner* o *Inductive Miner*;

- Conformance Checking: dati un *event log* ed un modello di processo, si verifica se le tracce contenute nel *log* rispettano il modello di processo; ciò è utile per verificare se le attività sono state eseguite secondo gli standard e le policy aziendali;
- Model Enhancement: dati un modello di processo ed un *event log*, si tenta di ottimizzare il modello;

In alcuni casi, i processi tendono ad essere fortemente complessi e variabili, perciò il modello ottenuto non è sempre comprensibile per l'operatore umano.
Per risolvere tale problematica, è possibile utilizzare tecniche di analisi di sotto-processi, dividendo il modello in componenti più piccole e facilmente interpretabili, oppure tecniche di compressione del processo, eliminando delle sotto-strutture in modo da rendere più comprensibile il processo nella sua interezza.
Nel seguente caso, prendendo spunto dall'algoritmo *Subdue* [1], si è realizzato un metodo per la compressione del processo basato sul concetto di *Local Process Models* (LPM).

1.2 Algoritmo Subdue

Subdue è un algoritmo di *clustering* gerarchico e concettuale, ne deriva che ogni *cluster* è descritto da un'etichetta e diviso in sotto-cluster.
L'obiettivo è di comprimere il grafo, sostituendo tutte le istanze di una sotto-struttura con un puntatore alla struttura stessa. Per determinare il sotto-grafo candidato, il criterio di scelta si basa sulla *Minimum Description Length* (MDL):

$$Compression = \frac{DL(S)+DL(G|S)}{DL(G)}$$

dove $DL(G)$ è la *Description Length* del grafo di input, $DL(S)$ è la *Description Length* della sotto-struttura e $DL(G|S)$ è la *Description Length* del grafo di input dal quale sono state eliminate tutte le istanze della sotto-struttura. La *Description Length* di un grafo si calcola come il numero di bit necessari per codificare la *matrice di adiacenza*, ovvero la struttura dati che implementa il grafo al calcolatore.

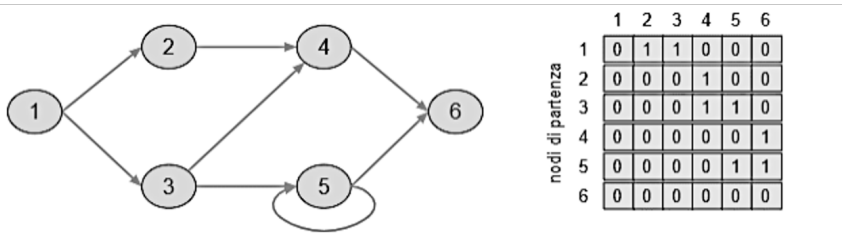


Figura 1.1: Grafo con la corrispondente matrice di adiacenza nodo-nodo.

La struttura scelta è quella che massimizza l'inverso di *Compression*.
Esistono diversi algoritmi in letteratura, dipendenti dal linguaggio di programmazione con cui sono stati implementati; nella seguente trattazione si farà riferimento alla versione codificata nel linguaggio Python, disponibile al seguente repository GitHub.
Il funzionamento dell'algoritmo può essere sintetizzato nei seguenti passi:

- Generazione dei sotto-grafi iniziali: per ogni arco del grafo, viene generato un nuovo grafo costituito dall'arco stesso e dai due nodi che esso collega; durante questa fase si verifica se i grafi generati sono isomorfi ad uno dei grafi generati precedentemente, ed in caso affermativo si aggiorna il numero di occorrenze del grafo già presente; successivamente queste strutture vengono inserite in una lista, all'interno della quale gli elementi sono ordinati in base al proprio valore di compressione in maniera decrescente.

- Ogni sotto-grafo nella lista viene esteso aggiungendo in maniera incrementale un arco; anche in questa fase si controlla se i grafi generati sono isomorfi ad uno dei grafi già presenti nella lista;

- Al termine delle fasi precedenti si ottiene una lista di sotto-strutture ordinate in base al valore di compressione e definite *grafi candidati*; perciò il grafo da sostituire è quello in testa.

- Il grafo viene sostituito con un nodo nominato $PATTERN\text{-}<NumeroIterazione>$.

- Il tutto viene ripetuto in base al numero di iterazioni definite dall'utente.

Mentre nell'algoritmo originale si vuole minimizzare *Compression*, nella seguente trattazione l'obiettivo è massimizzarla; tale incongruenza è dovuta al differente metodo di calcolo:

$$Compression = \frac{N(S) * A(S)}{A(G)}$$

dove $N(S)$ è il numero di istanze della sotto-struttura, $A(S)$ è il numero di archi della sotto-struttura e $A(G)$ è il numero di archi del grafo di input.

1.3 Obiettivi del progetto

L'obiettivo della seguente trattazione è la realizzazione un algoritmo di compressione che accetti in input *file di log* ed elimini delle sotto-strutture che soddisfino particolari proprietà.
L'approccio utilizzato è stato scomporre il problema principale in sotto-problemi discussi singolarmente, sintetizzandoli come segue:

- Discussione delle problematiche di adattamento dell'algoritmo *Subdue* relativamente alla seguente trattazione e discussione delle scelte adottate;
- Strumenti per la generazione dei *Local Process Model* (LPM) ed etichettatura del *file di log*;
- Progettazione dell'algoritmo;
- Sviluppo ed implementazione;
- Conclusioni e sviluppi futuri.

Capitolo 2

Problematiche dell'algoritmo Subdue

Come descritto nella sezione 1.2, *Subdue* consente di comprimere strutture a grafo, mentre nella seguente trattazione è necessario comprimere un *file di log*.
Per questo motivo, si renderebbero necessarie le seguenti modifiche all'algoritmo:

- La struttura in ingresso deve essere un *file di log*;
- Utilizzare una differente metrica di compressione che tenga conto del concetto di *traccia*;
- Il processo di sostituzione deve eliminare dal *file di log* gli eventi della sotto-struttura di interesse, sostituendo la prima occorrenza dell'evento con un'etichetta.

Alla luce delle precedenti problematiche, si è deciso di sviluppare un nuovo algoritmo indipendente da *Subdue*; il workflow della progettazione è rappresentato nella seguente figura:

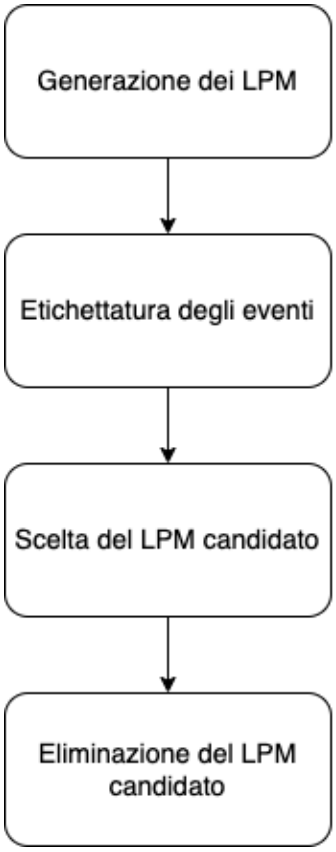


Figura 2.1: Workflow.

- Nella sezione 3.1 si tratterà della generazione dei LPM attraverso il framework ProM;
- Nella sezione 3.2 si applicherà l'algoritmo di etichettatura degli eventi del Log in base ai LPM in cui essi compaiono;
- Nella sezione 3.3 si discuterà della scelta della struttura candidata che, se eliminata, comprimerà *l'event log*;
- Nell'ultima fase (sezione 3.4) si elimina il candidato dal file di log.

Capitolo 3

Progettazione

3.1 Generazione dei LPM tramite ProM

ProM è un *framework* open-source sviluppato in Java, estensibile e capace di supportare una grande varietà di tecniche di *Process Mining* sotto forma di *plug-in*. Nella seguente trattazione, è stato utilizzato per generare i *Local Process Model* (LPM), i quali sono analoghi ai *grafi candidati* generati da *Subdue*.
Un *Local Process Model* (LPM) è una *rete di Petri* di piccole dimensioni, generalmente composta da cinque o sei nodi, che identifica un comportamento molto frequente in un *event log*; Tramite ProM, sono stati generati i LPM che fungeranno da input per l'algoritmo di etichettatura degli eventi nel *file di log*.

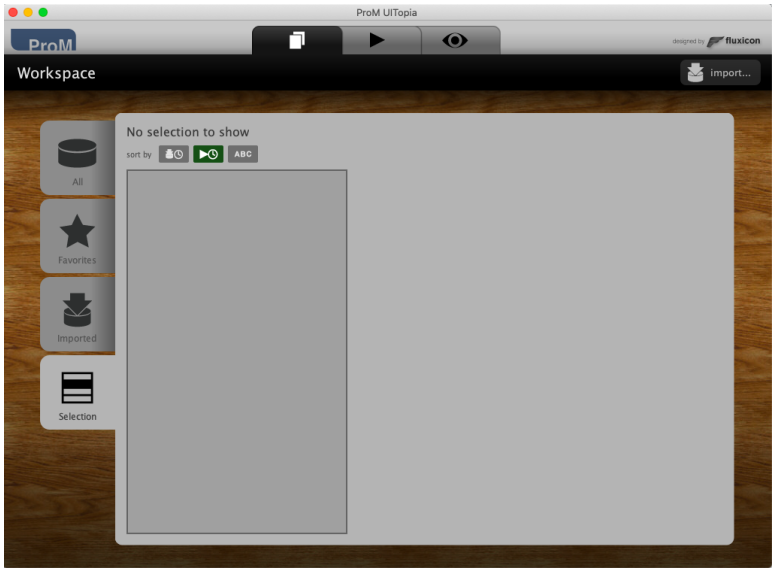
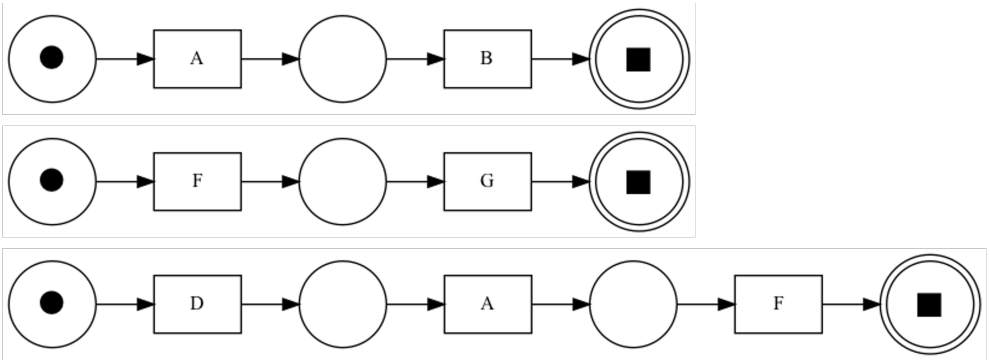


Figura 3.1: Schermata di ProM.

3.2 Etichettatura del file di log

L'algoritmo di etichettatura,fornito dal tutor di progetto Prof.ssa Genga Laura, richiede in input un *file di log* in formato *.xes* ed i LPM precedentemente ottenuti. In output, restituisce il file *.xes* originale in cui ad ogni evento è assegnata una lista che contiene, se presenti, gli indici dei LPM in cui l'evento compare.
L'evento, per poter essere etichettato, deve comparire in una *traccia* valida, altrimenti non viene contrassegnato. Una *traccia* si dice valida se percorre interamente il LPM dallo stato iniziale a quello finale, senza che rimangano *token* inutilizzati e senza aggiungerne dei nuovi.
Di seguito si riporta un esempio:
$L = [< a, b, c, d, a, b >, < a, c, d, a >, < f, g, h, i >, < d, a, f, g >]$



$L = [< a(1), b(1), c, d, a(1), b(1) >, < a, c, d, a >, < f(2), g(2), h, i >,$
$< d(3), a(3), f(2,3), g(2) >]$

3.3 Scelta del LPM più frequente

Per poter sostituire gli eventi nel *file di log*, è necessario definire una metrica per identificare il LPM che massimizza la compressione. Per la seguente trattazione, comprimere un *file di log* equivale ad eliminare il maggior numero di eventi possibili. Le metriche possibili sono:

- numero di eventi contrassegnati con il LPM: calcolata come il numero di eventi in cui occorre l'indice relativo al LPM;
- numero di tracce che percorrono il LPM moltiplicato per il numero di transizioni del LPM; è il metodo che maggiormente si avvicina alla metrica utilizzata da *Subdue* e si calcola come:

$$Ntracce * DimensioneLPM$$

L'algoritmo definito nella sezione 3.2 restituisce il file *.xes* originale in cui ad ogni evento è assegnata una lista che contiene, se presenti, gli indici dei LPM in cui l'evento compare in una traccia valida; perciò, il calcolo della seconda metrica è più semplice, poiché consiste nello scorrere l'*event log* e calcolare il numero di occorrenze di ogni LPM. Inoltre, tale metrica definisce esattamente il numero di eventi che saranno sostituiti, perciò si adatta meglio all'obiettivo del progetto.

3.4 Eliminazione del LPM più frequente

Dopo aver individuato il LPM candidato, si comprime il *file di log* modificando o eliminando gli eventi. Per far ciò, verrà utilizzata una variabile booleana per determinare se se l'evento corrente è il primo di una traccia valida per il LPM candidato, in questo caso viene sostituito da un evento con nome *LPM:<numLPM>_Iteration<NumIterazione>*; altrimenti viene eliminato. Inoltre, sarà necessario eliminare in ogni evento gli attributi *LPMs_list*, *LPMs_binary* e *LPMs_frequency*, perché ogni evento verrà etichettato nuovamente nella successiva iterazione.

Capitolo 4

Implementazione e Testing

4.1 Implementazione

L'implementazione completa dell'algoritmo, inclusi i successivi file utilizzati per il testing, sono disponibili pubblicamente nel seguente repository GitHub.

4.2 Pseudocodice

La seguente sezione contiene la logica esclusivamente relativa all'algoritmo di compressione, ovvero il modulo sviluppato per il seguente progetto didattico.

```
def next_transitions(net, transition):
    # searching places target from None transition
    for arc in list(transition.out_arcs):
            # searching all transitions out from the current
               place
            for arc in list(place.out_arcs):
                if target transition is not None:
                    # appending to list of initial markings
                else:
                    # recursion with same net without the
                       current place
```

```
def prev_transitions(net, transition):
    # searching places source from None transition
    for arc in list(transition.in_arcs):
         # searching all transitions in from the current place
        for arc in list(place.in_arcs):
            if source transition is not None:
                # appending to list of final markings
            else:
```

```
                # recursion with same net without the current
                    place
```

```
def extract_initial_final_markings(input_lpms, num):
    # there are two nets, because recursion for
        initial_markings delete places, so we risk that results
         of final_markings are influenced by the previous net.

    # reading pnml file

    # setting initial and final places

    #iterating on arcs' net, here we can iterate over net1 or
        net2, no differences
    for arc in list(next_net.arcs):
        if place == initial place:
            if label target is None:
                # must check next transitions
                next_transitions(next_net, target)
                # recursion whit net1

        if place == final_place:
            if label source is None:
                # must check previous transitions
            else:
                prev_transitions(prev_net, source)
                # recursion whit net2

    return ordered_initial_markings, ordered_final_markings
```

```
    def compression(input_xes, input_lpms, limit, out_xes,
        prefix, suffix):
    """
    Count number of occurrences in xes file, and each time that
        LPM appears, increasing relative index on the list
    """
    # reading xes file

    # extracting number of files contained in the LPMs folder

    # define a list that contains as many zeros as there are
        LPMs files

    # calculating most frequent index of LPMs
    iteration = 0
```

```
while(iteration < limit):
    for trace in file_xes:
        for event in trace:
            # getting LPM list
            for lpm in lpms_event_list:
                # updating count_list

    # getting most frequent index of LPMs

    # getting initial_markings and final_markings lists
    initial_markings, final_markings =
       extract_initial_final_markings(input_lpms,
       max_index)

    """
    Having found the LPMS that compresses the most, one
       modifies the xes by scrolling through the traces,
       and for each event,
    you check whether the index to be deleted appears in
       the LPMs attribute; if so,
    if the event is the first in the trace, the name is
       modified, otherwise the
    the event itself
    """
    for trace in file_xes:
        # index represents the index of the event on which
           it is iterating
        index = 0

        new_trace = True
        prev = None

        while(index < len(trace)):
            # setting the current event
            # setting the next event equal to the current
               +1

            # we get the list of LPMs

            if the index that appears multiple times is
               contained in the current lpm:
            if(max_index in lpms_event_list):

                if the the current event name appears in
                   final markings,and the next exists, and
```

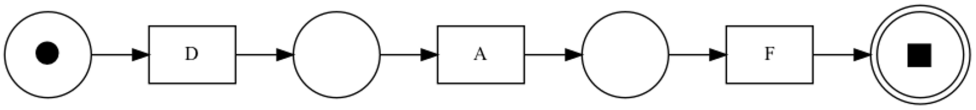
```
                    its name appears in initial markings:
                # new trace is starting
                    new_trace = True

                if a previous event exists, and its name is
                    equal to the current event, and the
                   initial_markings is equal to the final
                   markings:
                # it's a loop, so a new trace is must start
                    new_trace = True

                # setting the previous event equal to the
                    current one

                # event name is replaced, otherwise the
                    event itself is deleted.
                if(not new_trace):
                    # you delete the event
                else:
                    new_trace = False
                    # you replace the name and delete lpms
                        attributes.

                    # index is incremented to iterate on
                        the next event
            else:
                if the index of the most frequent does not
                   appear in the list of lpms:
                    # delete lpms attributes.
                new_trace = True

                # prev must be none here
                # index is incremented to iterate on the
                    next event
        # iteration is incremented

    # overwrite file if already exists
```

4.3 Testing

La componente testata è esclusivamente quella legata allo sviluppo del modulo di compressione, ovvero il file *compression_with_lpm.py* sviluppato per il seguente progetto didattico.
Viene dato in input all'algoritmo di compressione un file *.xes* in cui ad ogni evento

è assegnata una lista che contiene, se presenti, gli indici dei LPM in cui l'evento occorre.
L'algoritmo è stato testato su molteplici casistiche, di seguito ne vengono riportate alcune:

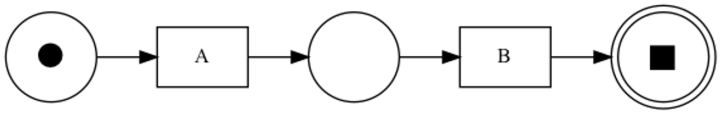
$$L = [< d(2), a(2), f(1,2), g(1), d(2), a(2), f(1,2) >]$$



$$Initial = [D],\ final = [F]$$
$$L = [< LPM: 2\_Iteration: 0, g, LPM: 2\_Iteration: 0 >]$$

Figura 4.1: Esempio di applicazione dell'algoritmo eliminando il LPM 2.

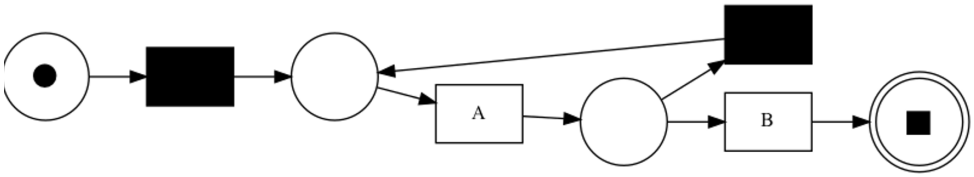
$$L = [< a(1), b(1), a(1), b(1), c, a(1), b(1) >]$$



$$Initial = [A],\ final = [B]$$
$$, = [< LPM: 1\_Iteration: 0, LPM: 1\_Iteration: 0, c, LPM: 1\_Iteration: 0 >]$$

Figura 4.2: Esempio di applicazione dell'algoritmo eliminando il LPM 1.

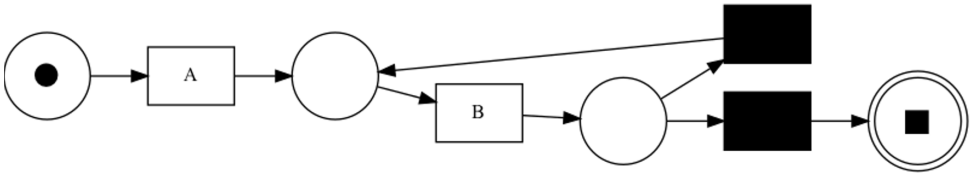
$$L = [< a(1), b(1), a(1), b(1), c, a(1), b(1) >]$$



$$Initial = [A],\ final = [B]$$
$$L = [< LPM: 1\_Iteration: 0, LPM: 1\_Iteration: 0, c, LPM: 1\_Iteration: 0 >]$$

Figura 4.3: Esempio di applicazione dell'algoritmo eliminando il LPM 1.

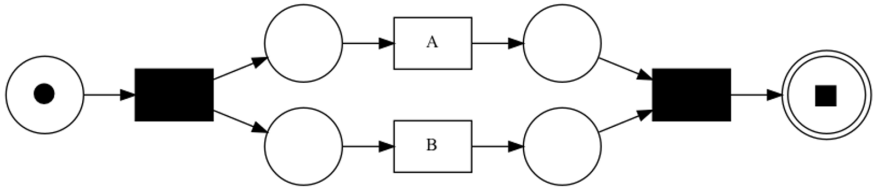
$$L = [< a(1), b(1), a(1), b(1), c, a(1), b(1) >]$$



$$Initial = [A],\ final = [B]$$
$$L = [< LPM{:}1\_Iteration{:}0, LPM{:}1\_Iteration{:}0, c, LPM{:}1\_Iteration{:}0 >]$$

Figura 4.4: Esempio di applicazione dell'algoritmo eliminando il LPM 1.

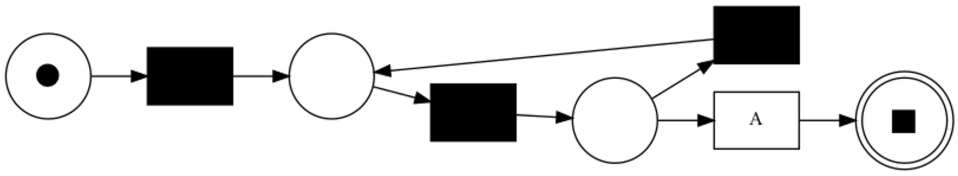
$$L = [< a(1), c, a(1), a(1) >]$$



$$Initial = [A,B],\ final = [A,B]$$
$$L = [< LPM{:}1\_Iteration{:}0, c, LPM{:}1\_Iteration{:}0, LPM{:}1\_Iteration{:}0 >]$$

Figura 4.5: Esempio di applicazione dell'algoritmo eliminando il LPM 1.

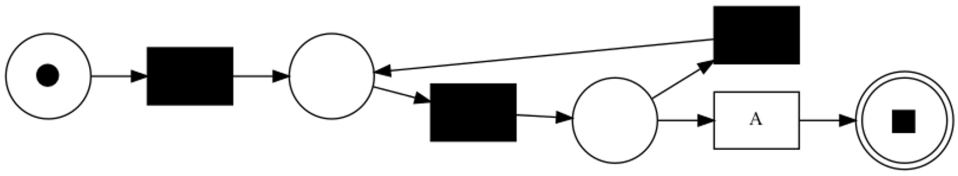
$$L = [< a(1), c, a(1), a(1) >]$$



$$Initial = [A],\ final = [A]$$
$$L = [< LPM{:}1\_Iteration{:}0, c, LPM{:}1\_Iteration{:}0, LPM{:}1\_Iteration{:}0 >]$$

Figura 4.6: Esempio di applicazione dell'algoritmo eliminando il LPM 1.

$$L = [< a(1), b(1), b(1), b(1), c, a(1), b(1) >]$$



$$Initial = [A], final = [A]$$
$$L = [< LPM: 1\_Iteration: 0, c, LPM: 1\_Iteration: 0 >]$$

Figura 4.7: Esempio di applicazione dell'algoritmo eliminando il LPM 1.

Capitolo 5

Conclusioni e sviluppi futuri

L'obiettivo della seguente trattazione, come precedentemente discusso, è di realizzare un algoritmo di compressione che operi con la stessa logica di *Subdue*, accettando in input *file di log* anziché grafi.
Alcuni sviluppi futuri potrebbero coinvolgere:

- Ottimizzazione dell'algoritmo di etichettatura;
- Realizzazione di un programma che inglobi le fasi di generazione dei LPM (al momento realizzato separatamente tramite interfaccia grafica dal framework ProM) ed il qui proposto algoritmo di compressione.

Bibliografia

[1] Istvan Jonyer, Diane J Cook, and Lawrence B Holder. Graph-based hierarchical conceptual clustering. *Journal of Machine Learning Research*, 2(Oct):19–43, 2001.

Elenco delle figure