UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Algoritmo BIG: Implementazione in Python

Big Data Analytics and Machine Learning

Autori:

Cappanera Simone S1102490
Ciaffoni Massimo S1102853
Thaliath Amal Benson S1101080

Referente: Prof. Potena Domenico
Co-Referenti: Genga Laura
Anno accademico: 2020-21

Indice

Indice **i**

Introduzione **1**

1 Struttura ed implementazione dell'algoritmo **3**

1.1 Tecnologie utilizzate 4

1.2 Strutture dati utilizzate 4

1.3 Find Causal Relationship 4

1.4 Extract Instance Graph 5

1.5 Check Trace Conformance 7

1.6 Irregular Graph Repairing 8

1.6.1 Deletion Repair 9

1.6.2 Insertion Repair 11

1.6.3 File di output 13

1.6.4 Note implementative 13

2 Risultati ottenuti **14**

3 Conclusioni e sviluppi futuri **21**

Bibliografia **22**

Introduzione

Il *Process Mining* è una disciplina che ha lo scopo di scoprire, monitorare e migliorare un dato processo sfruttando il log degli eventi generato durante l'esecuzione del processo. In particolare una tecnica di questa disciplina è il *Process Discovery* che viene utilizzato per ottenere la rappresentazione sottoforma di rete del modello di processo tramite l'analisi di un log del processo contenente la sequenza delle varie attività. Tra gli algoritmi più utilizzati nel Process Discovery ci sono *l'alpha miner* il quale analizza le relazioni tra le attività all'interno delle tracce (footprint) e ne estrae un modello e *l'inductive miner*, il quale invece analizza le frequenze delle relazioni tra le varie attività delle tracce per estrarne successivamente un modello. Il modello di rete estratto può essere valutato tramite un task di *Conformance Checking* il quale analizza una ad una le tracce presenti nel log effettuando un aligment tra la traccia e il modello.

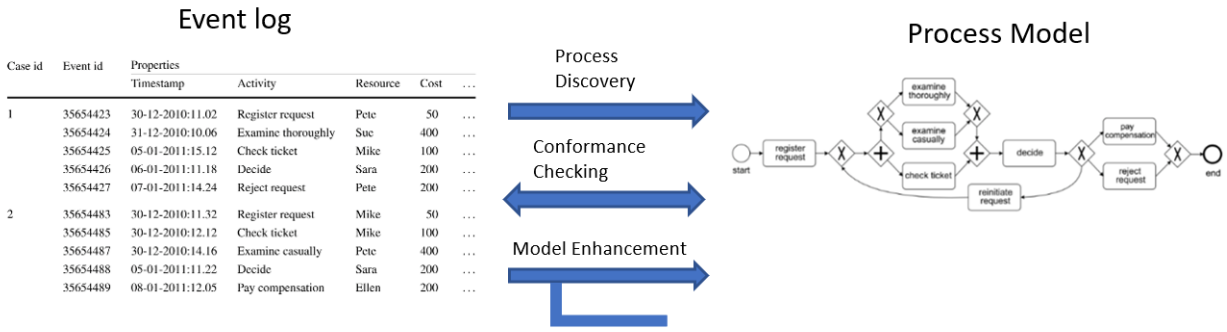


Figura 1: Fasi del Process Mining

In molti casi il modello di processo non è strettamente necessario per le analisi, mentre una rappresentazione della singola istanza di processo può fornire anche essa molte informazioni e mostrare parallelismi nascosti nella traccia sequenziale. Questi modelli di istanza sono rappresentati sottoforma di *Instance Graph*.

Building Instance Graphs

Un algoritmo che sia in grado di estrarre questi Instance Graph da modelli molto complessi, detti *spaghetti-like*, è descritto nel paper *Building instance graphs for highly variable processes* [1]. I modelli in questione sono spesso caratterizzati da elevata variabilità nella sequenza delle varie attività; le quali non permetteno l'utilizzo delle classiche tecniche di Process Mining (utili invece per modelli strutturati).
L'obiettivo di questo progetto è produrre un'implementazione in Python di tale algoritmo. L'algoritmo BIG è utilizzato per generare ed eventualmente riparare i grafi di processi molto complessi in modo da estrarre succesivamente tramite l'uso di approcci di subprocess mining i vari comportamenti locali del modello o analizzare le diverse anomalie presenti.
Il codice è reperibile a:
`https://colab.research.google.com/drive/1VOMBTSBwqSlHVaP2k4Npy2kx20ZJd_fG?usp=sharing`

1 | Struttura ed implementazione dell'algoritmo

L'algoritmo è implementato come una combinazioni di funzioni più piccole che eseguono ciascuna le parti in pseudo-codice riportati nel paper.
L'algoritmo riceve in input il file di log e il file della rete di petri (i path dei file), degli offset per indicare la traccia di inizio e fine del log, e un parametro opzionale per dire se visualizzare in output i vari instance graph. In output vengono prodotti diversi file testuali per ciascun instance graph riparato.

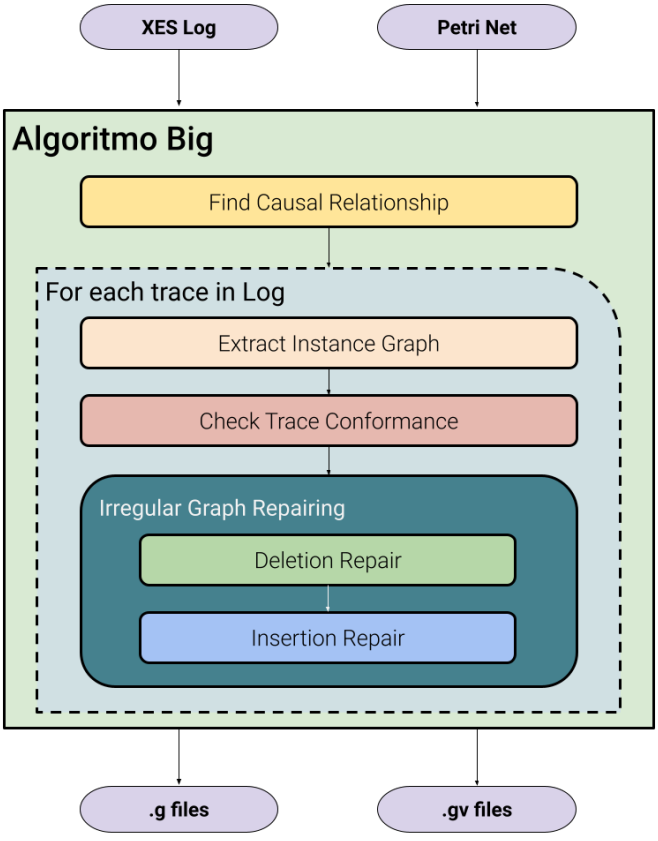


Figura 1.1: Struttura generale dell'algoritmo.

1.1. Tecnologie utilizzate

L'algoritmo è stato implementato in *Python 3.7*, la versione con più compatibilità al momento della stesura di questa relazione. In particolare sono stati utilizzati:

- **Google Colaboratory:** Servizio di Google messo a disposizione per sviluppare ed eseguire notebook Jupiter di Python in modo condiviso in cloud.
- **PM4PY:** [2] Libreria Python dedicata al process mining. Contiene diversi moduli per il graph mining, conformance checking e molto altro. La versione utilizzata per il progetto è la *2.2.15*.

1.2. Strutture dati utilizzate

Per questa particolare implementazione i nodi degli instance graphs sono rappresentati tramite tuple Python `(ID, LABEL)` dove il primo elemento è un id univoco e il secondo l'etichetta dell'attività associata all'evento.
Gli archi diretti sono rappresentati come tuple `(E1, E2)` dove i membri sono i nodi rappresentati come precedentemente descritto e la direzione dell'arco va dal primo al secondo evento.
Infine ogni instance graph è descritto da una lista Python di nodi ed una di archi. Verranno spesso riferiti rispettivamente come `V` e `W`.

1.3. Find Causal Relationship

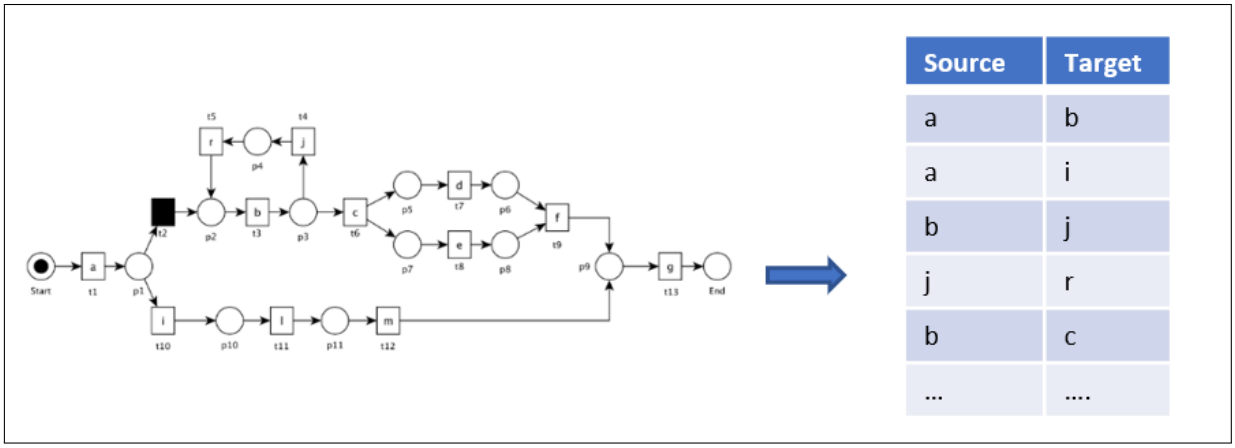


Figura 1.2: Esempio di estrazione di Casual Relation

Questa funzione si occupa di estrarre le *causal relationship* del modello di processo. Prende in input la rete di petri e gli indicatori dei place iniziale e finale. Restituisce in output una lista di coppie dove la seconda attività segue direttamente la prima. Queste relazioni sono calcolate con l'apposita funzione PM4PY `footprints_discovery.apply(net, im, fm)`.

Tenere ben presente che se alcune causal relationship non vengono rilevate evidentemente la rete non è *sound*.

```
1 from pm4py.algo.discovery.footprints
2     import algorithm as footprints_discovery
3
4 def findCausalRelationships(net, im, fm):
5   fp_net = footprints_discovery.apply(net, im, fm)
6   return list(fp_net.get('sequence'))
```

1.4. Extract Instance Graph

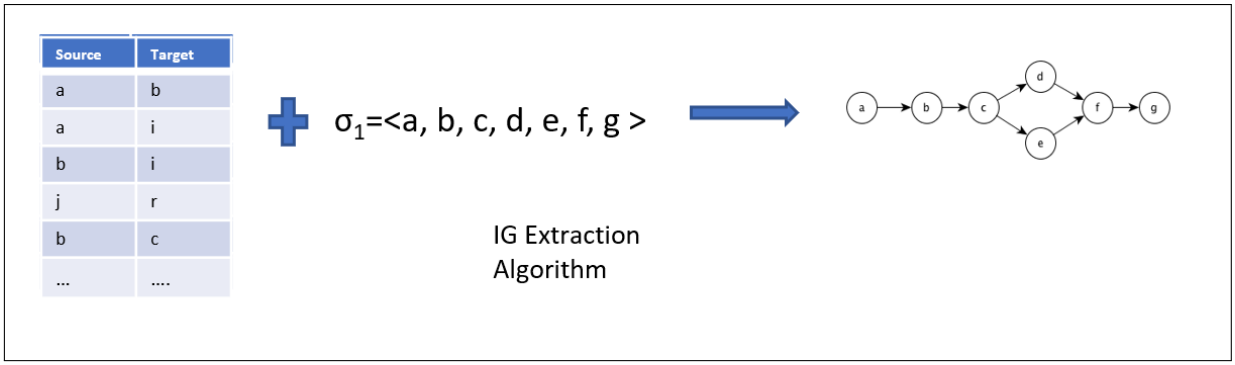


Figura 1.3: Esempio di estrazione di un grafo

Questa funzione, preso in input una traccia e la lista delle causal relationship, produce in output l'instance graph rappresentato da una lista di nodi e una di archi. La funzione è basata sulla definizione 18 del paper di riferimento.

In breve una volta creato un nodo per ogni evento della traccia, con id incrementale in base alla posizione dell'evento nella traccia, viene creato un arco per ogni coppia di eventi (e_1, e_2) se: e_2 segue e_1 nella traccia, se le attività associate sono in relazione causale $a_1 \underset{CR}{\longrightarrow} a_2$, e non sono presenti uno o più eventi intermedi fra e_1, e_2 tale per cui $a_1 \underset{CR}{\longrightarrow} a_j$ e $a_k \underset{CR}{\longrightarrow} a_2$.

```
def ExtractInstanceGraph(trace, cr):
    V = []
    W = []
    id = 1
    for event in trace:
      V.append((id, event.get("concept:name")))
      id += 1
    for i in range(len(V)):
      for k in range(i+1,len(V)):
        e1 = V[i]
        e2 = V[k]
        if (e1[1],e2[1]) in cr:
          flag_e1=True
          for s in range(i+1, k):
            e3 = V[s]
            if (e1[1],e3[1]) in cr:
              flag_e1 = False
              break
          flag_e2=True
          for s in range(i+1, k):
            e3 = V[s]
            if (e3[1],e2[1]) in cr:
              flag_e2 = False
              break

          if flag_e1 or flag_e2:
            W.append((e1,e2))
    return V, W
```

Nel caso in cui nella traccia sia presente un'attività di tipo inserted o deleted l'algoritmo estrarrà un grafo irregolare come mostrato nella figura seguente.

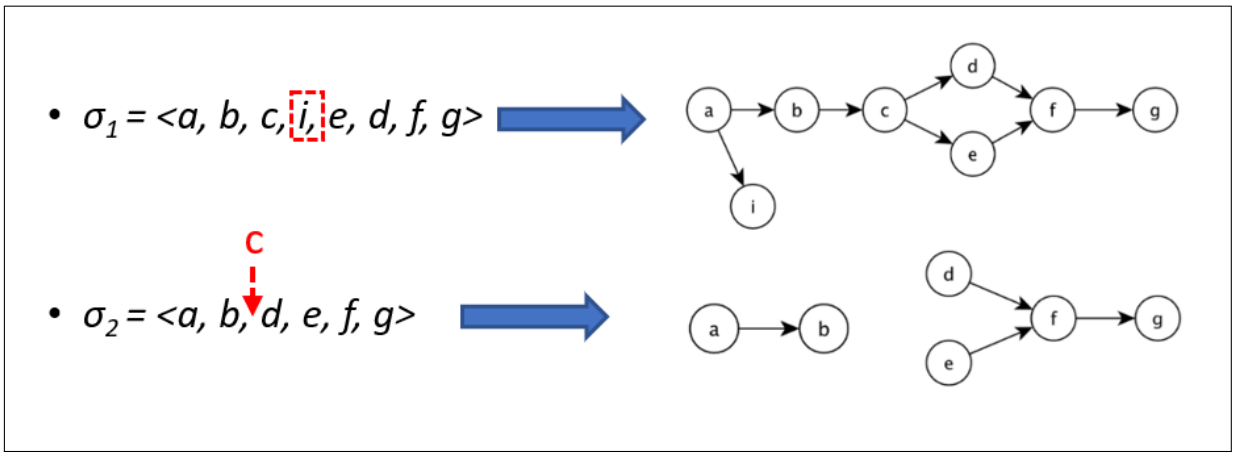


Figura 1.4: Esempio di estrazione di grafi irregolari

1.5. Check Trace Conformance

Questa funzione controlla se una determinata traccia è conforme a una rete di Petri tramite il conformance checking basato sull'allineamento. Essa prende in input la traccia, la rete di Petri e gli indicatori dei place iniziale e finale. Restituisce due liste: una lista di sequenze delle attività eliminate e una lista di sequenze delle attività inserite nella traccia. Ogni elemento delle due liste è una lista essa stessa. Questa sottolista è formata da eventi in successione dove gli elementi sono coppie dove il primo membro è la posizione dell'attività cancellata/inserita e il secondo è l'etichetta dell'attività. Nelle sequenze delle attività cancellate il valore della posizione è sempre lo stesso, mentre per le attività inserite è incrementale. Questa funzione è stata scritta sulla base della sola descrizione della natura delle liste di attività cancellate/inserite senza riferimenti a nessun tipo di pseudo-codice. L'allineamento è stato ottenuto tramite una funzione apposita di PM4PY che utilizza di default l'algoritmo A*.

```
1  from pm4py.algo.conformance.alignments.petri_net
2      import algorithm as alignments
3
4  def checkTraceConformance(trace, net, initial_marking,
5                            final_marking):
6
7    aligned_traces = alignments.apply_trace(trace, net,
8                              initial_marking, final_marking)
9    D = []
10   I = []
11   id = 0
12   temp_d = []
13   temp_i = []
14   prev_d = False
15   curr_d = False
16   prev_i = False
17   curr_i = False
18   del_count = 1
19   for edge in aligned_traces['alignment']:
20     id+=1
21     if edge[1] is None:
22       id-=1
23       continue
24     if edge[0] == '>>':
25       temp_d.append((id, edge[1]))
26       curr_d = True
27       id-=1
28     if edge[1] == '>>':
29       temp_i.append((id, edge[0]))
30       curr_i = True
31
32     if (prev_i and not curr_i):
33       if len(temp_i) > 0:
34         I.append(temp_i)
```

```
35        temp_i = []
36      prev_i = curr_i
37      curr_i = False
38      if (prev_d and not curr_d):
39        if len(temp_d) > 0:
40          D.append(temp_d)
41        temp_d = []
42
43      prev_d = curr_d
44      curr_d = False
45    if len(temp_i) > 0:
46        I.append(temp_i)
47    if len(temp_d) > 0:
48        D.append(temp_d)
49    return D, I
```

1.6. Irregular Graph Repairing

Questa e le successive funzioni sono la caratteristica principale dell'algoritmo e si occupano appunto di riparare gli instance graphs costruiti su tracce non conformi. La funzione riceve in input le liste di nodi ed archi dell'instance graph, le liste di attività cancellate ed inserite e le causal relationships.

All'interno vengono create due liste contenenti una le etichette di tutte le attività cancellate e una con tutti gli eventi inseriti (quindi sia id che attività). Queste verranno passate rispettivamente al deletion repair e all'insertion repair che vengono richiamate per ciascuna sottolista di attività cancellata/inserita in successione.

L'output della funzione è l'instance graph riparato (nuova lista di archi `Wi`) e un oggetto graphviz per salvataggi successivi.

```
1  def irregularGraphRepairing(V, W, D, I, cr, view=False):
2    Wi=W
3    all_deleted_labels = []
4    for d_element in D:
5        for element in d_element:
6          if element[1] not in all_deleted_labels:
7            all_deleted_labels.append(element[1])
8    for d_element in D:
9      Wi=DeletionRepair(Wi, V, d_element,cr, all_deleted_labels)
10   print("Deletion␣repaired␣Instance␣Graph")
11   graph = viewInstanceGraph(V, Wi, view)
12
13   all_inserted = []
14   for i_element in I:
15     for i in i_element:
16       if i not in all_inserted:
17         all_inserted.append(i)
18   for i_elements in I:
19     Wi=InsertionRepair(Wi,V,i_elements,cr, all_inserted)
```

```
20    print("Insertion repaired Instance Graph")
21    graph = viewInstanceGraph(V, Wi, view)
22    return Wi, graph
```

1.6.1. Deletion Repair

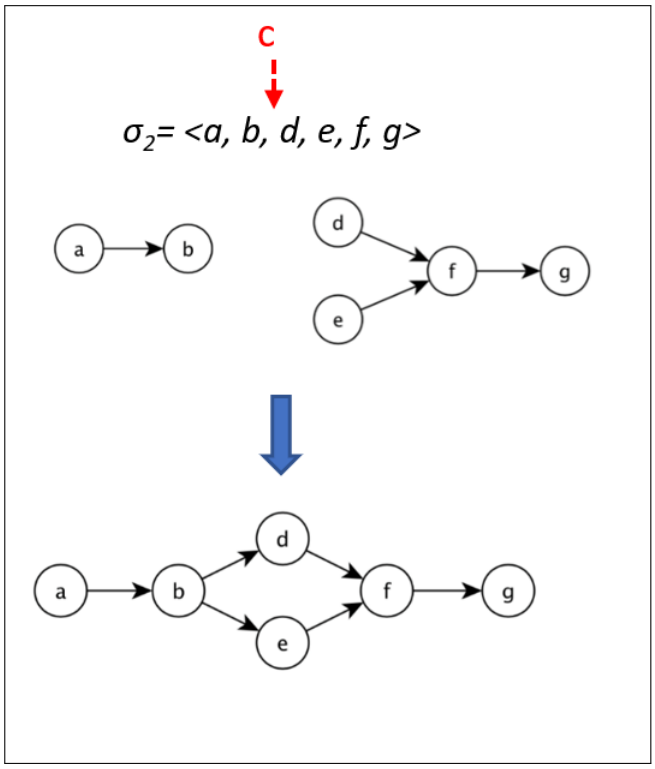


Figura 1.5: Esempio di deletion repair

In poche parole la Deletion Repair non fa altro che eliminare gli archi creati erroneamente durante la generazione dell'instance graph per via delle attività mancanti (righe 7-26) e creare nuovi archi per rimuovere le discontinuità create al passo precedente (righe 27-43). Per una spiegazione più dettagliata ed esaustiva fare riferimento al sotto-capitolo 5.1 del paper di riferimento.

```
1  def DeletionRepair(Wi, V, d_elements, cr, all_deleted):
2    v_len = len(V)
3    Wr1 = []
4    Wr2 = []
5    i = d_elements[0][0]
6
7    if i <= v_len:
8      for edge in Wi:
9        if edge[1][0] == i and edge[0][0] < i
10       and (d_elements[-1][1],V[i-1][1]) in cr:
11         for h in range(edge[0][0], i):
```

```
12            if (V[h-1][1],d_elements[0][1]) in cr:
13              Wr1.append(edge)
14              break
15
16        if edge[0][0] < i and edge[1][0] > i
17        and (d_elements[-1][1],edge[1][1]) in cr:
18          if edge[0][1] in all_deleted:
19            Wr2.append(edge)
20          elif (edge[0][1],d_elements[0][1])  in cr:
21            for l in range(i+1, edge[1][0]):
22              if (V[l-1],edge[1]) in Wi:
23                Wr2.append(edge)
24                break
25
26    Wi = list(set(Wi) - set(Wr1 + Wr2))
27    for k in range(i - 1, 0, -1):
28      for j in range(i, v_len+1):
29        if (V[k-1][1],d_elements[0][1]) in cr:
30          if (d_elements[-1][1], V[j-1][1]) in cr:
31            if not isReachable(V, Wi, V[k-1], V[j-1]):
32              flag1 = True
33              for l in range(k + 1, j):
34                if (V[k-1],V[l-1]) in Wi:
35                  flag1 = False
36                  break
37              flag2 = True
38              for m in range(k + 1, i):
39                if (V[m-1],V[j-1]) in Wi:
40                  flag2 = False
41                  break
42              if flag1 or flag2:
43                Wi.append((V[k-1],V[j-1]))
44    return Wi
```

1.6.2. Insertion Repair

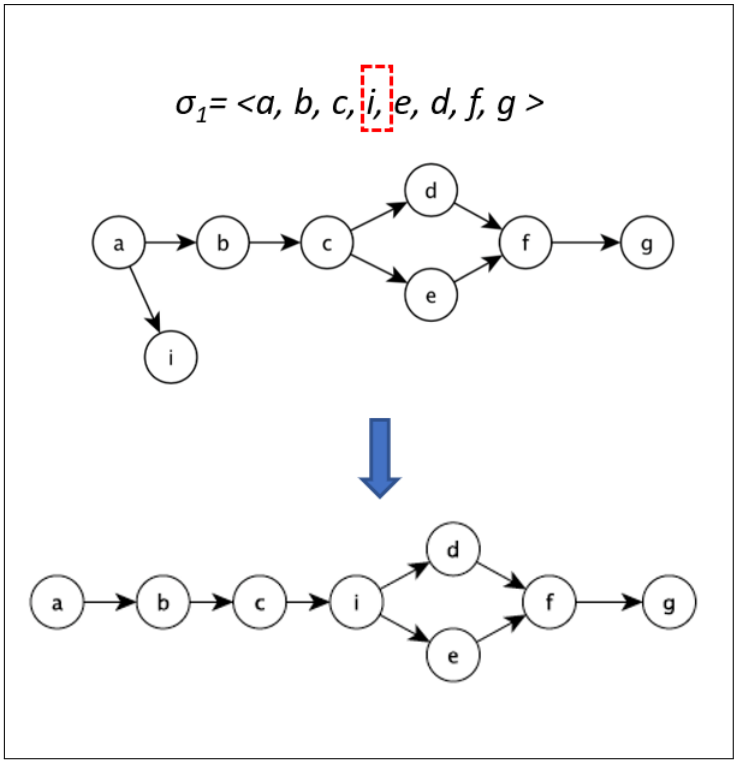


Figura 1.6: Esempio di insertion repair

L'Insertion Repair agisce in maniera simile alla deletion repair. Prima di tutto va a rimuovere gli archi tra gli eventi inseriti e quelli precedenti ad essi, tra gli eventi inseriti e i successivi, ed infine gli archi interni fra gli inseriti stessi (righe 15-23). Di seguito vengono riparate le discontinuità collegando gli eventi precedenti con il primo evento inserito e gli eventi successivi con l'ultimo (righe 25-43). Allo stesso modo vengono collegati, fra quelli inseriti, ogni evento con il proprio successivo (righe 45-48). Infine un'ultima sezione si occupa di eliminare eventuali archi ridondanti ancora presenti (righe 50-62).
Per una spiegazione più dettagliata ed esaustiva fare riferimento al sotto-capitolo 5.2 del paper di riferimento.

```
1  def InsertionRepair(W, V, i_elements, cr, all_inserted):
2    v_len = len(V)
3    Wr1=[]
4    Wr2=[]
5    Wr3=[]
6    Wr4=[]
7    Wr5=[]
8    Wa1=[]
9    Wa2=[]
10   Wa3=[]
```

```
11    i= i_elements[0][0]
12    j=i+len(i_elements)-1
13    Wi=W.copy()
14
15    for edge in Wi:
16      if edge[0][0]<i and edge[1][0]>=i and edge[1][0]<=j:
17        Wr1.append(edge)
18      if edge[0][0]>=i and edge[0][0]<=j and edge[1][0]>j:
19        Wr2.append(edge)
20      if edge[0][0]>=i and edge[0][0]<=j
21      and edge[1][0]>=i and edge[1][0]<=j:
22        Wr3.append(edge)
23    Wi= list(set(Wi) - set(Wr1 + Wr2 + Wr3))
24
25    for k in range(j+1, v_len+1):
26        if V[k-1] not in all_inserted:
27          if (V[i-2][1],V[k-1][1]) in cr or (V[i-2],V[k-1]) in Wi:
28            if not isReachable(V, Wi, V[j-1], V[k-1]):
29              Wi.append((V[j-1],V[k-1]))
30              Wa1.append((V[j-1],V[k-1]))
31
32    # if i < v_len and (V[i-2][1],V[i][1]) not in cr:
33    if i == v_len or (V[i-2][1],V[i][1]) not in cr:
34        Wi.append((V[i-2],V[i-1]))
35        Wa2.append((V[i-2],V[i-1]))
36    else:
37      for k in range(i-1,0,-1):
38          if V[k-1] not in all_inserted:
39            if j < v_len and ((V[k-1][1],V[j][1]) in cr
40            or (V[k-1],V[j]) in Wi):
41              if not isReachable(V, Wi, V[k-1],V[i-1]):
42                Wi.append((V[k-1],V[i-1]))
43                Wa2.append((V[k-1],V[i-1]))
44
45    for k in range(i, j):
46      Wa3.append((V[k-1],V[k]))
47    if len(Wa3)>0:
48        Wi=Wi+Wa3
49
50    for edge in Wa2:
51      for edge2 in Wa1:
52        if edge[1][0]>=i and edge[1][0]<=j:
53          if edge2[0][0]>=i and edge2[0][0]<=j:
54            Wr4.append((edge[0],edge2[1]))
55    Wi= list(set(Wi) - set(Wr4))
56
57    # if i < v_len and (V[i-2][1],V[i][1]) not in cr
58    if i == v_len or (V[i-2][1],V[i][1]) not in cr:
59      for edge in Wi:
60        if edge[1][0]>i and edge[0][0]==i-1:
61          Wr5.append(edge)
62      Wi = list(set(Wi) - set(Wr5))
63    return Wi
```

1.6.3. File di output

L'algoritmo crea principalmente tre tipi di file di output.
Innanzitutto per ogni traccia viene salvato sia un file .gv, formato default degli oggetti graphviz, e un file .g dove sono riportati: tramite commenti (denotati con '#') il tempo di esecuzione, le attività inserite e le attività cancellate; i nodi dell'instance graph (denotati con 'v') con id ed etichetta dell'attività; gli archi dell'instance graph (denotati con 'e') con id del primo e secondo evento e le rispettive etichette separate da due underscore __.
Infine viene creato un file .txt generale che riporta il tempo di esecuzione totale, il numero di tracce totali e quelle riparate.

1.6.4. Note implementative

Più volte nello pseudo codice presente nel paper si definiscono insiemi di archi (e_i, e_j) che soddisfano un certo insieme di condizioni e soprattutto che appartengano all'insieme di archi dell'instance graph. L'implementazione più efficiente è creare questi insiemi ciclando su ciascun arco dell'instance graph e verificando che gli indici dei nodi dell'arco soddisfino le restanti condizioni. Vedere l'esempio seguente:

$W_{r1} = \{(e_k, e_i) \mid k < i \wedge (e_k, e_i) \in W \ldots\}$

```
1 for edge in W:
2   if edge[1][0] == i and edge[0][0] < i
3   ...
4     ...
5       ...
6         Wr1.append(edge)
```

Si può constatare infatti che la traduzione più diretta, ovvero generare tutte le coppie di nodi possibili e verificare che appartengano all'insieme di archi W, è molto più onerosa computazionalmente perché nel caso medio per ogni coppia si dovrebbe comunque andare a ciclare nella lista W per verificarne l'appartenenza.

Altra importante questione da tenere in mente è che il conformance checking tramite alignment della libreria PM4PY non è deterministico nelle tracce in cui è presente un *move-in-log* e un *move-in-model* in sequenza. In questo caso infatti l'ordine di questi due move-in varia da esecuzione a esecuzione, e ciò influisce sull'indice associato alle attività nelle liste di attività inserite ed eliminate (vedere sezione 1.5). Tuttavia questo problema non influisce sul corretto funzionamento dell'algoritmo e il risultato finale è equivalente, ma potrebbe causare confusione.

2 | Risultati ottenuti

L'efficacia generale dell'algoritmo è valutata nel paper originale. Per quanto riguarda l'efficienza di questa implementazione sono riportati di seguito alcune statistiche su dei modelli di esempio, tuttavia va tenuto presente che l'ottimizzazione del codice non era fra gli obiettivi principali di questa prima versione in Python.
Il primo esempio riguarda il log e la rete del modello *BPI2012DecompositionExpr*.
I risultati sono i seguenti:

- **Numero di tracce:** 7455
- **Tracce irregolari:** 4144
- **Tempo totale:** 356,702 s
- **Tempo medio:** 0,048 s

Di seguito sono riportati la rete di Petri e un esempio di instance graph derivato da traccia irregolare e le relative riparazioni.

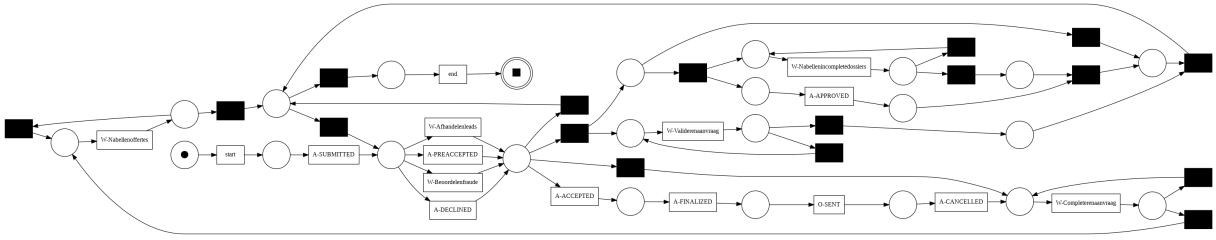


Figura 2.1: Rete di Petri del processo BPI 2012

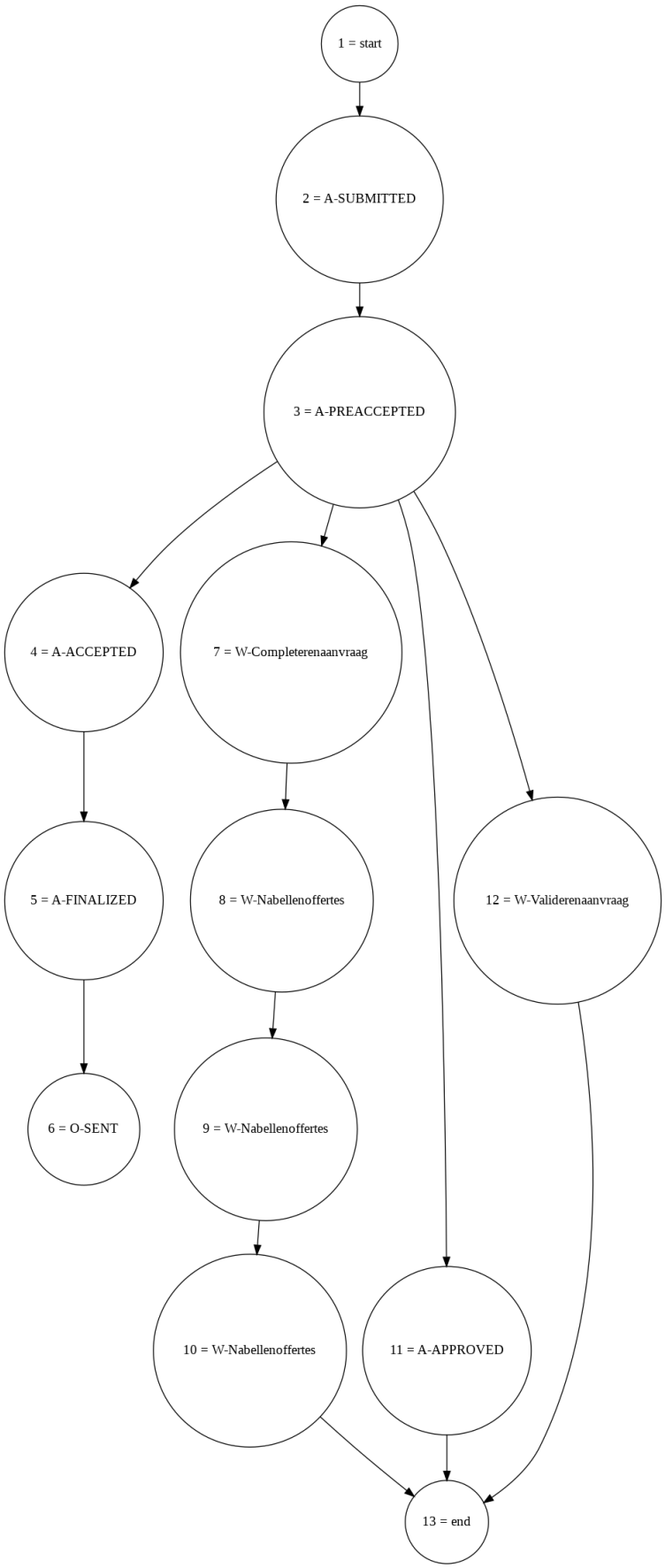
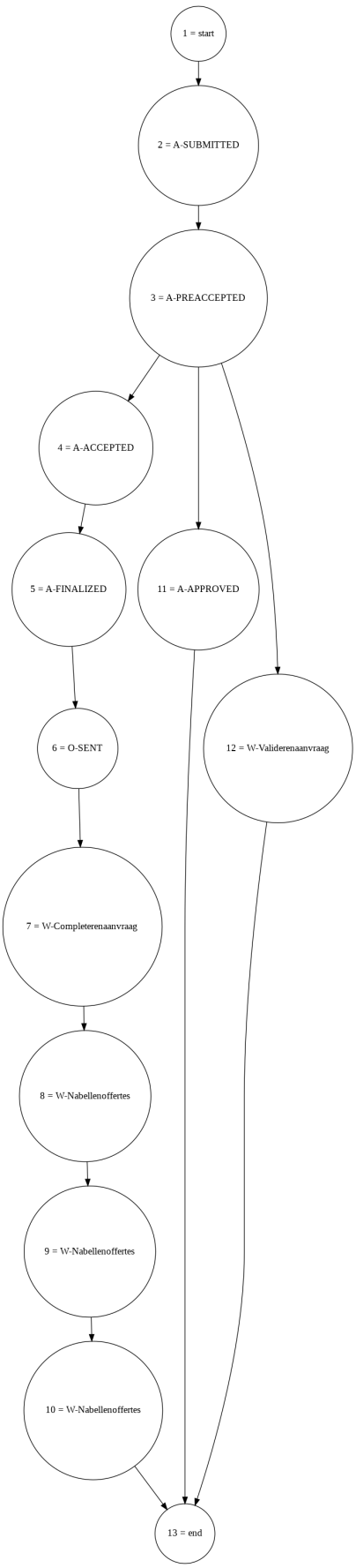
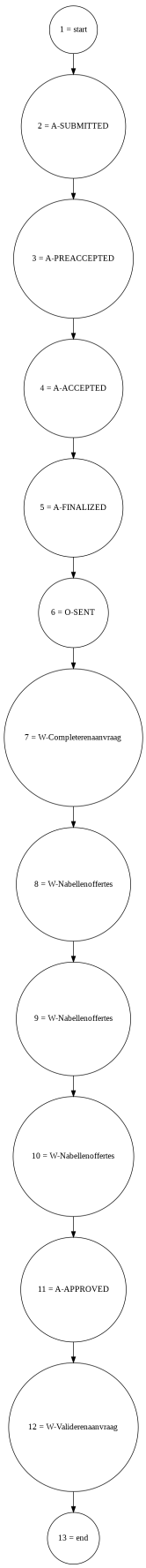


Figura 2.2: Instance graph non riparato



(a) Deletion Repair



(b) Insertion Repair

Un altro esempio riguarda il modello *TestBank2000NoRandomNoise*. I risultati sono i seguenti:

- **Numero di tracce:** 1500
- **Tracce irregolari:** 1500
- **Tempo totale:** 74,341 s
- **Tempo medio:** 0,05 s

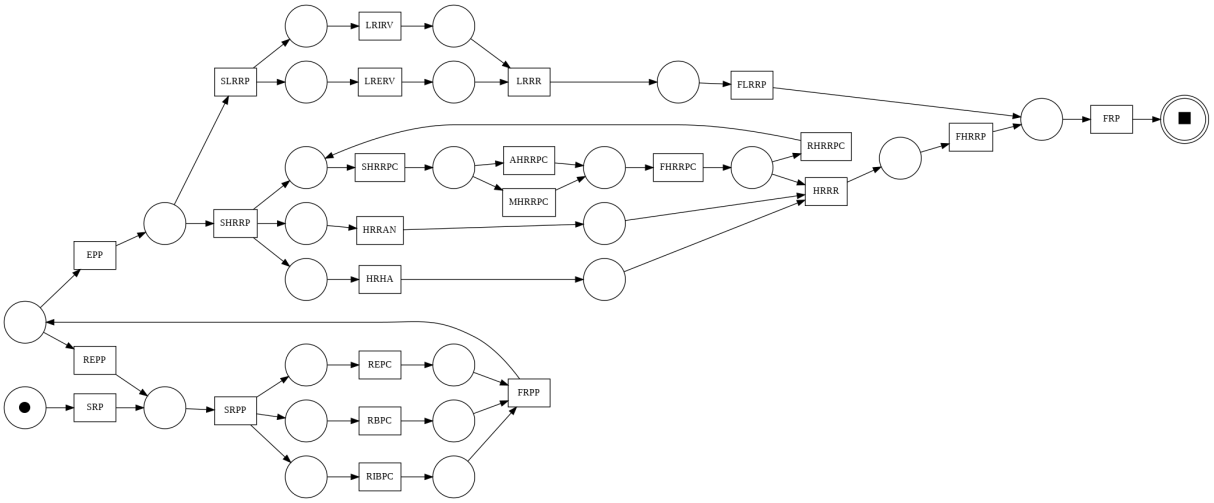


Figura 2.3: Rete di Petri del processo Test Bank

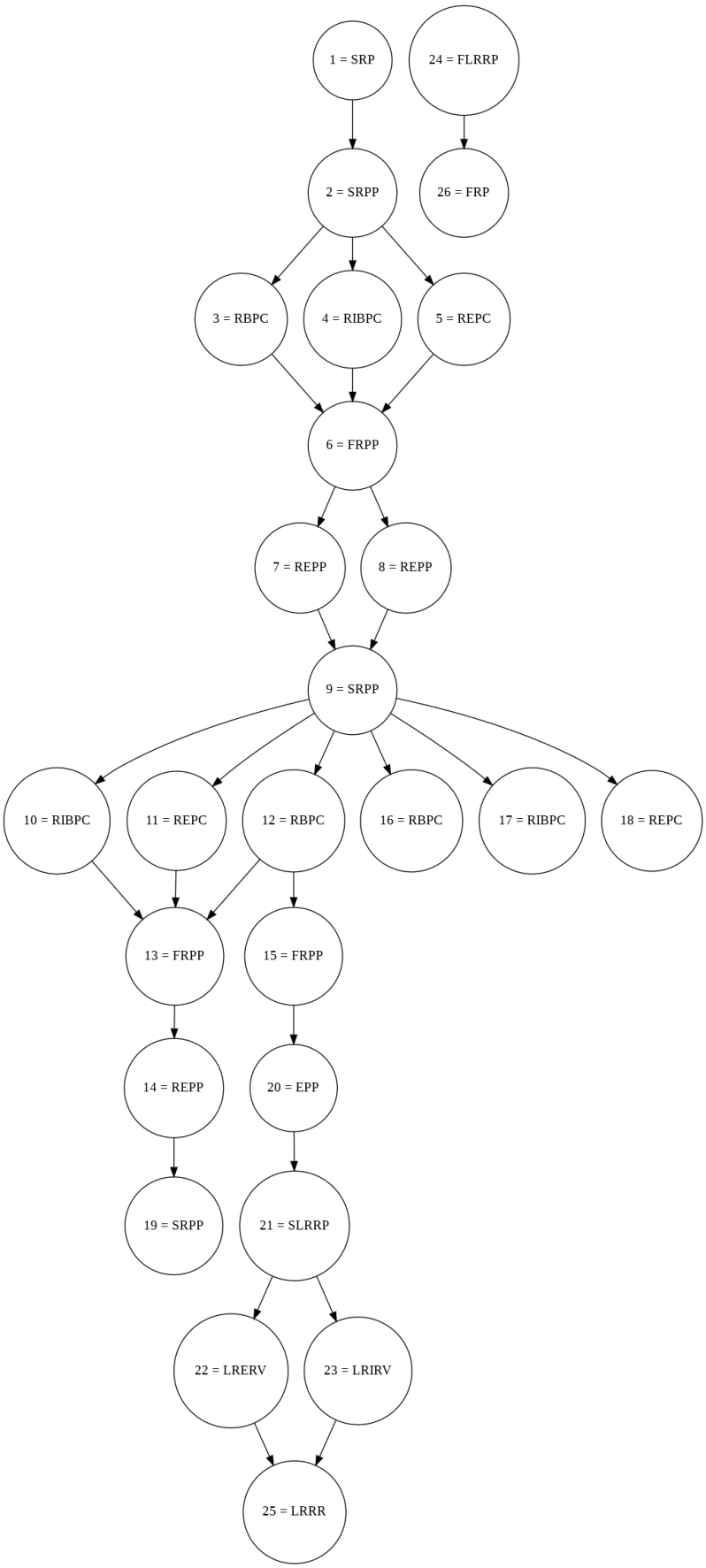
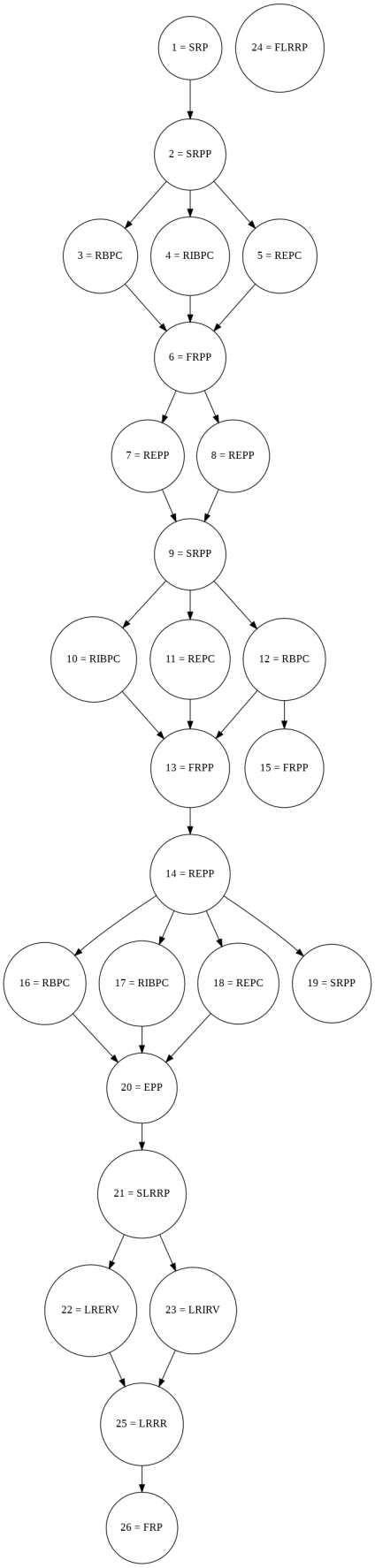
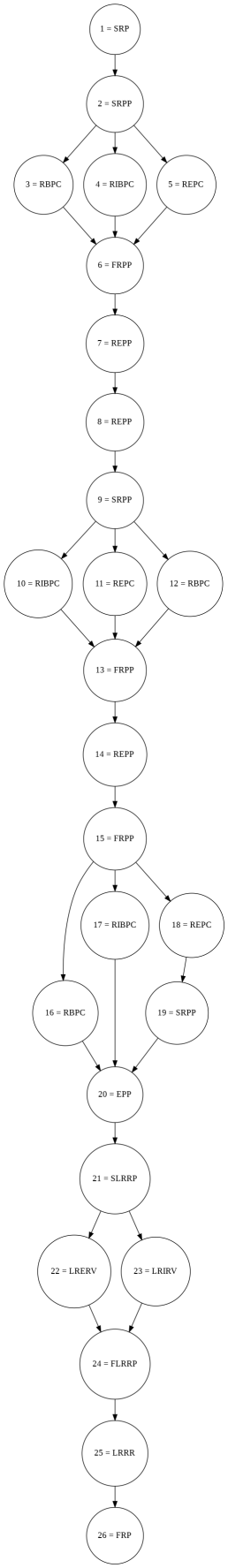


Figura 2.4: Instance graph non riparato



(a) Deletion Repair



(b) Insertion Repair

Di seguito sono riportati i risultati di altri due esempi di log e modello.

- *BPI Denied*
  - **Numero di tracce:** 3093
  - **Tracce irregolari:** 3093
  - **Tempo totale:** 803,342 s
  - **Tempo medio:** 0,26 s
- *TestBank2000SCC*
  - **Numero di tracce:** 1995
  - **Tracce irregolari:** 1995
  - **Tempo totale:** 84,582 s
  - **Tempo medio:** 0,042 s

3 | Conclusioni e sviluppi futuri

L'implementazione risulta funzionare correttamente ed è conforme ai risultati ottenuti dalla precedente implementazione in Java. Oltre che all'introduzione di eventuali aggiornamenti del paper, principale obiettivo tra gli sviluppi futuri è migliorare l'ottimizzazione del codice attraverso il calcolo distribuito tramite *PySpark*, ristrutturazione del codice, ed un uso più massiccio di librerie come Numpy per velocizzare i calcoli su array/liste.

Bibliografia

[1] C. Diamantini, L. Genga, D. Potena, and W. V. D. Aalst. Building instance graphs for highly variable processes. *Expert Systems with Applications*, 59:101–118, 2016. doi: 10.1016/j.eswa.2016.04.021.

[2] F. I. for Applied Information Technology. Pm4py, state-of-the-art-process mining in python. URL `https://pm4py.fit.fraunhofer.de/`.