UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTA' DI INGEGNERIA

Corso di Laurea magistrale in Ingegneria Informatica e dell'Automazione

PROGETTAZIONE DI UNA METAEURISTICA PER L'ESTRAZIONE DI INSTANCE GRAPHS DA TRACCE NON CONFORMI

DESIGNING A METAHEURISTIC FOR INSTANCE GRAPHS DISCOVERY FROM NON-CONFORMANT TRACES

Relatore:

Prof. Domenico Potena

Correlatrice:

Prof.ssa Ornella Pisacane

Tesi di Laurea di:

Martina Pioli

Anno Accademico 2020 / 2021

A Silvana e Marco,

che l'hanno reso possibile

Indice

Introduzione **5**

1 Definizioni preliminari **9**

2 Building Instance Graph **15**

2.1 Limitazioni 17

3 Soluzione proposta **18**

3.1 Iterated Local Search 18

3.2 Funzione obiettivo 19

3.3 Outline della metaeuristica proposta 21

3.4 Simulated Annealing 22

3.5 Pseudocodice 25

3.5.1 Main 26

3.5.2 Riparare la soundness dell'IG 28

3.5.3 Local Search 30

3.5.4 Accettazione dell'IG 31

4 Sperimentazione **33**

4.1 Mini dataset 33

4.2 Esempio esteso 37

4.2.1 Dataset 37
4.2.2 Scelta dei parametri 38
4.2.3 Algoritmi a confronto 39
4.2.4 IG a confronto 40

5 Conclusioni **44**

Introduzione

Oggigiorno le organizzazioni utilizzano software per supportare l'esecuzione di processi. Il Business Process Management System (BPMS), i file system condivisi, email e messaggistica sono tutti strumenti che tengono anche traccia delle attività svolte dagli utenti, cioè delle esecuzioni di istanze di processo.

Una traccia è composta da una sequenza di eventi i quali, in base al sistema utilizzato, sono a loro volta composti da un elenco di attributi. Quelli più comuni sono: tempo d'inizio dell'esecuzione, identificatore dell'attività, risorsa che l'ha eseguita e tempo di termine o durata totale.
La raccolta dei dati di tutte le tracce, che vengono di volta in volta eseguite nell'organizzazione, confluisce in un documento: l'**event log**.
Il valore ottenuto da questi dati è inestimabile per le aziende che, attraverso gli analisti, riescono ad avere informazioni riguardo i comportamenti interni: possono monitorare le performance operative ed elaborare delle strategie per migliorarle.

È possibile immaginare la traccia come una lista di eventi avvenuti in modo sequenziale tra loro: al termine di un evento ne inizia un altro. Da questo tipo di sequenze possono essere fatte delle semplici analisi che portano ad ottenere informazioni utili come ad esempio il tempo medio di esecuzione di un'attività, l'utilizzo delle risorse ed i tempi morti tra due eventi successivi.

Pensando però a tracce reali, svolte all'interno di organizzazioni, possono esserci anche due o più attività che vengono eseguite in parallelo ovvero in modo indipendente tra loro: devono essere eseguite al termine di un'attività dal quale il parallelismo parte e tutte devono terminare, senza altri vincoli di successione, prima di poter proseguire l'esecuzione della traccia nella successiva attività dopo il parallelismo.
Un modello d'istanza, mappato ad esempio in un **Instance Graph** (IG), riesce a rappresentare esplicitamente questo tipo di esecuzione, che rimarrebbe altrimenti nascosto nelle tracce sequenziali, rendendolo disponibile per analisi aggiuntive con lo scopo di ottenere altre importanti informazioni.

Per costruire degli IG è necessario conoscere le relazioni causali tra le attività, che possono essere dedotte da un modello o da un event log. In questo documento è trattata la costruzione di IG derivanti da event log semplici, con i soli attributi base: l'identificatore e l'etichetta di ogni evento. Non verranno presi in considerazione i tempi di esecuzione né le risorse coinvolte nel processo.

La disciplina che si occupa del rilevamento delle relazioni tra le attività a partire da un event log è il Process Discovery (PD), esso rende possibile l'estrazione del modello d'istanza basandosi sulle regole indotte dalle relazioni causali implementate nella tecnica dell' α-algorithm [vdA16]. Tale tecnica però ha delle limitazioni: opera abbastanza bene se si hanno tracce di processi ben strutturati, cioè che non si discostano (o si discostano poco) dal modello; non riesce ad ottenere dei buoni risultati quando viene applicato a tracce *altamente variabili* ovvero che si allontanano molto dal comportamento standard.

I casi reali che si vogliono considerare in questo documento avranno delle tracce altamente variabili tipiche del comportamento umano. Infatti, se vengono inseriti attori umani nello svolgimento dei processi, sarà presente un alto grado di libertà nell'esecuzione delle attività. Per rendersi conto della differenza basta pensare alle diverse problematiche e casistiche che possono presentarsi nello svolgimento di una procedura ospedaliera rispetto ad una procedura industriale: è possibile analizzare il processo di entrambe, ma nella prima ci sarà una probabilità molto più alta che qualche attività si discosti dal modello standard previsto.

La limitazione dell'applicazione dell'α-algorithm è stata studiata in letteratura ed è relativa alla costruzione di modelli affetti da overgeneralizzazione, ovvero un modello che genera un elevato numero di attività che possono essere eseguite in parallelo e che permettono molte più istanze di processo rispetto a quelle realmente ammissibili.
Per migliorare la qualità degli IG estratti, in termini di accuratezza e minor generalizzazione, si potrebbero utilizzare delle tecniche di filtraggio che permettano di modellare solo i comportamenti di processo più frequenti.
Tuttavia, questa tecnica porta a scartare i comportamenti irregolari rispetto al modello perché possono essere considerati *sbagliati* o *anomali*. Questo non è accettabile nel dominio dei processi altamente variabili: le tracce che si discostano dal modello potrebbero essere la fonte d'informazione più importante.
Valutare un comportamento irregolare può essere utile in due casi: per intervenire internamente all'organizzazione ed evitare che quella deviazione sia permessa; analizzare l'informazione sotto un occhio diverso per raggiungere un risultato molto più importante. Le risorse umane facenti parte del processo acquisiscono esperienza nel dominio e potrebbero aver trovato un comportamento che si discosta dal modello che sia però più efficiente o più ottimizzato. In questo secondo caso si potrà valutare la modifica del processo standard per migliorarlo a livello di organizzazione.

In letteratura esiste un algoritmo capace di creare degli Instance Graph per processi altamente variabili [DGPvdA16] che però presenta delle limitazioni. Nel presente elaborato ci si pone l'obiettivo di utilizzare una metaeuristica per superarle ed ottenere degli instance graph che rispecchino il modello di processo iniziale e che abbiano un livello di generalizzazione il più basso possibile.

Il contenuto del documento si articola nelle seguenti sezioni:

- nel primo capitolo verranno riportate delle definizioni propedeutiche per la comprensione dell'elaborato;
- nel secondo capitolo verrà analizzata la soluzione esistente: l'algoritmo Building Instance Graph (BIG);
- l'analisi continuerà con una discussione sulle limitazioni di tale soluzione;
- nel terzo capitolo verrà presentata la soluzione alternativa basata sulla metaeuristica con particolare attenzione alla funzione obiettivo utilizzata;
- attraverso degli esempi e dei dati concreti, nel quarto capitolo, si vuole mostrare il risultato ottenuto.

Capitolo 1

Definizioni preliminari

Questa sezione introduce degli importanti concetti che verranno usati nel documento richiamando delle definizioni di base della letteratura del Process Mining [vdA16].

Modello di Processo Un *modello di processo* è un insieme di regole che definiscono lo svolgimento di un insieme di attività che ne fanno parte.

Petri Net Una *Petri net*, o rete di Petri, è formata da:

- un insieme P di places;
- un insieme T di transizioni;
- un insieme $F \subseteq (P \times T) \cup (T \times P)$ per indicare il flusso delle relazioni tra le attività.

Nella figura 1.1 è possibile vedere un esempio di modello di processo, rappresentato attraverso una Petri net. Si noti come le attività vengano mappate nell'insieme delle transizioni nei quadrati, i places sono invece identificati dai cerchi.

Tale modello sarà di riferimento per il resto del documento.

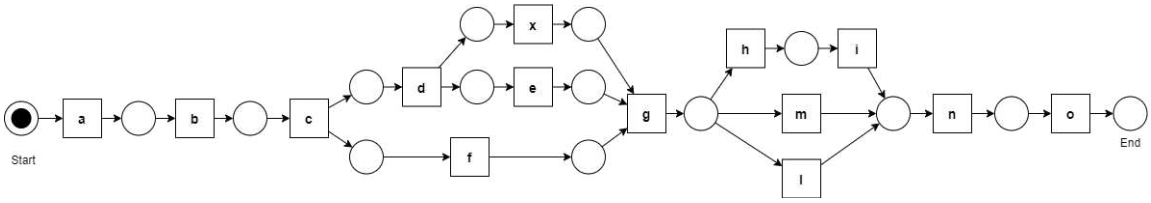

Figura 1.1: Il modello di processo visualizzato con una Petri net

Event Log Un *event log* è un insieme di tracce.
L'obiettivo del Process Discovery è quello di ottenere un modello di processo a partire da un event log dato.

Attività Un'*attività* è un task che può essere eseguito nel processo.
$A = \langle a, b, c, d, e, f, x, g, h, i, l, m, n, o \rangle$ è l'insieme di attività riferito all'esempio del modello di processo.
Siano $a_1, a_2 \subseteq A$ due attività, una coppia (a_1, a_2) può essere:

- in sequenza $\rightarrow$ se per iniziare a_2 deve terminare prima a_1. Nelle coppie (b, c), (b, d) l'ordine di esecuzione delle attività deve corrispondere a quello della coppia stessa.
- in parallelo $\rightarrow$ se le due attività possono essere svolte in qualsiasi ordine ovvero sono indipendenti tra loro. Un esempio sono le coppie (d, f), (x, e) che possiamo trovarle in una traccia sia nell'ordine mostrato che in quello inverso.
- alternative $\rightarrow$ se non possono essere eseguite entrambe ovvero si può percorrere un solo ramo. Nel nostro modello di processo la coppia (l, m) è alternativa.

Traccia Una *traccia* è la mappatura di un'istanza di processo, ovvero una specifica esecuzione del modello di processo, nella rispettiva sequenza di attività che si è verificata. Ogni elemento della traccia è un *evento*, ovvero una specifica istanza d'attività.

È possibile trovare un'attività ripetuta più volte nella stessa traccia, ognuna corrisponderà ad un evento diverso. Tuttavia, in questo progetto, il modello di processo sarà aciclico omettendo la possibilità che si verifichi una situazione del genere. Per questo possiamo utilizzare la semplificazione che ci permette di identificare gli eventi con le etichette delle attività alle quali sono riferite. Nel caso in cui il modello dia la possibilità di ripetizione di determinate attività è necessario etichettare gli eventi di una traccia aggiungendo un identificatore univoco.
Viene di seguito riportato un esempio di traccia:

$$\sigma_1 = \langle a, b, c, d, e, f, x, g, m, n, o \rangle$$

Conformità Per *conformità* s'intende la completa corrispondenza di una traccia al modello di processo. In altre parole: prendendo la traccia, evento per evento, sarà possibile percorrere la Petri net associata al modello, senza dover saltare nessuna attività o trovare eventi corrispondenti ad attività aggiuntive. In caso contrario la traccia si dice *non conforme*.
Prendendo sempre come modello di riferimento quello in fig. 1.1:

- σ_1 è conforme;
- $\sigma_2 = \langle a, b, d, f, e, x, g, l, m, n, o \rangle$ non è conforme. Nella traccia manca l'evento c e gli eventi l ed m sono alternativi, non possono essere eseguiti entrambi come in questo caso.

Criteri di qualità di un modello Per valutare la bontà di un modello di processo si cerca di trovare un trade-off tra i seguenti quattro criteri:

- **Fitness**: il modello deve permettere di replicare i comportamenti dai quali è stato estratto. In altre parole, ogni traccia dell'event log deve essere conforme al modello;
- **Precisione**: il modello ricavato non deve permettere istanze di processo completamente scollegate da quelle dell'event log;
- **Generalizzazione**: il modello deve generalizzare i comportamenti dell'event log;
- **Semplicità**: il modello deve essere il più semplice possibile.

Tali criteri sono stati presi in considerazione per lo svolgimento di questo progetto e verranno ripresi più volte nel documento.

Relazioni Causali Una *relazione causale* (CR) è una relazione che rappresenta l'ordine di esecuzione tra una coppia di attività di un processo. Ad esempio, siano $a_1, a_2 \subseteq A$, scrivere $a_1 \rightarrow_{CR} a_2$ indica che a_2 non può iniziare l'esecuzione fino a che a_1 non sia terminata; in altre parole l'esecuzione di a_2 (*causal successor*) dipende da quella di a_1 (*causal predecessor*).

Instance Graph Un *instance graph* (IG) è un grafo che rappresenta il flusso delle attività di una specifica istanza di processo, o traccia.

È definito con la coppia di due insiemi nodi-archi (V, W).

L'insieme V è formato da un nodo per ogni evento della traccia, si ha una mappatura uno-a-uno, ognuno dei quali è etichettato con l'attività corrispondente.
L'insieme W è formato dagli archi che permettono d'indicare l'ordine degli eventi della traccia, facendo riferimento alle relazioni causali basate sul modello di processo di riferimento.

In figura 1.2 è riportato un esempio: $IG(\sigma_1)$, l'instant graph estratto dalla traccia σ_1, utilizzata precedentemente, a partire dalle CR del modello in fig. 1.1.

IG di tracce non conformi Capita di avere una traccia dell'event log non conforme con il modello di processo. In base alle situazioni si possono intraprendere due strade: scartare la traccia o considerarla un'informazione importante da analizzare.
Con l'idea che una traccia non conforme possa portare a delle analisi molto importanti, in questo progetto si vuole prendere la seconda via e lavorare proprio su questo tipo di IG. Utilizzando lo stesso algoritmo di estrazione si otterranno però dei grafi con delle caratteristiche diverse rispetto a quelle di un IG conforme.
In figura 1.3) si riporta $IG(\sigma_2)$.

Soundness Data la seguente definizione: un *nodo iniziale* è un nodo che non ha nessun arco in input, viceversa, un *nodo finale* non ha nessun arco in output; un grafo si dice *sound* se ha un solo nodo iniziale ed un solo nodo finale. In altre parole un grafo è sound se è connesso.

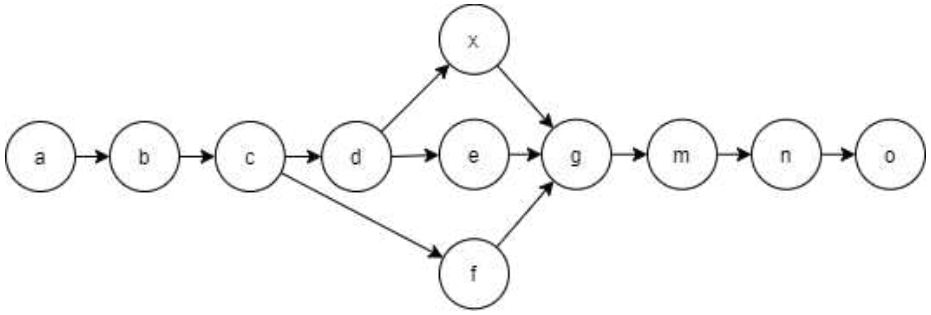

Figura 1.2: IG della traccia σ_1 conforme al modello

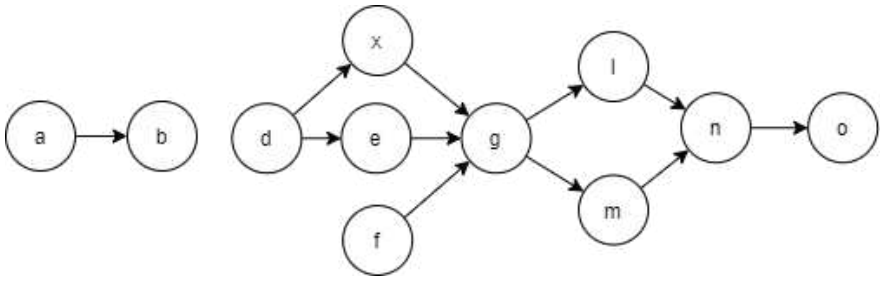

Figura 1.3: Instant Graph della traccia σ_2 non conforme al modello

Capitolo 2

Building Instance Graph

Lo scopo dell'algoritmo Building Instance Graph (BIG) è quello di costruire un buon modello d'istanza. Gli instance graph costruiti per processi altamente variabili, a partire da tecniche già conosciute, portano a risultati di scarsa qualità, per questo si aggiunge una seconda fase di riparazione che mira a migliorare l'IG creato inizialmente.

Estrazione del modello di processo

Esistono in letteratura diverse tecniche di PD per la derivazione di un modello di processo che descriva l'intero event log, ma gli approcci che permettono di derivare modelli per singole istanze di processo sono veramente pochi.
Di questi approcci abbiamo due categorie:

1. model-based: data una rete di Petri sono in grado di ricavare le possibili esecuzioni che corrispondono a tutti i percorsi ammissibili;

2. log-based: a partire da un event log, si derivano le relazioni causali tra le attività di processo che verranno poi utilizzate per costruire i modelli d'istanza.

Entrambi gli approcci sono affetti dalle limitazioni di cui abbiamo parlato nell'introduzione: restituiscono IG che overgeneralizzano. Tuttavia l'algoritmo BIG per ottenere un primo grafo dal quale partire utilizza l'approccio log-based, che ritroveremo anche nella metaeuristica.

Repairing

Lo scopo della seconda fase dell'algoritmo è quella di modificare l'IG per rappresentare meglio quelle tracce dell'event log che non sono conformi al modello di partenza. La procedura presenta delle somiglianze con l'idea proposta in [FvDA15] poiché entrambe sfruttano il conformance checking per rilevare le irregolarità da riparare. L'algoritmo però si differenzia da questa tecnica, è implementata una personalizzazione per analizzare e riparare i modelli a istanza singola.

Se la traccia non è conforme, in questa fase vengono identificati diversi tipi di evento. La categorizzazione viene fatta sulla base della tecnica dell'*alignment* utilizzando da una parte la traccia e dall'altra il modello di riferimento:

- *attività corrette*: evento della traccia al quale corrisponde un'attività nel modello;
- *attività inserite*: evento della traccia al quale non corrisponde un'attività nel modello, chiamato anche *move on log*;
- *attività eliminate*: attività nel modello alla quale non corrisponde un'attività nella traccia, chiamato anche *move on model*.

L'algoritmo restituisce quindi due liste con gli eventi che si trovano nelle ultime due categorie. A partire da quegli eventi vengono richiamati due sub-algoritmi che permettono di riparare queste anomalie fino ad ottenere un IG corretto e che sia possibile analizzare successivamente.

2.1 Limitazioni

Questo algoritmo presenta delle limitazioni dovute all'approccio rigido con il quale effettua le riparazioni. I due sub-algoritmi che riparano le attività inserite e quelle eliminate operano secondo delle precise regole.

Come spiegato nell'introduzione, si stanno effettuando riparazioni per tracce altamente variabili, le quali possono allontanarsi anche di molto dal modello di processo al quale fanno riferimento. Questa caratteristica porta ad un numero molto elevato di casistiche di fronte alle quali ci si può trovare. In altre parole, le tracce possono presentare anomalie di diversi tipi, combinandosi in un modo non considerato al momento della progettazione dell'algoritmo. Di conseguenza, non sono considerate tutte le possibili situazioni nelle regole di riparazione.

Un'altra limitazione del Building Instance Graph, dovuta sempre alla rigidità delle regole applicate, è quella di ottenere un unico risultato anche se un IG di una traccia non conforme può essere riparato in modi diversi ma ugualmente ammissibili.
Ad esempio, l'attività inserita m dell'IG in fig. 1.3, verrebbe riparato da BIG mettendolo in sequenza alle altre attività, come previsto dalla traccia. Ma non possiamo dire a priori che quella sia la soluzione più corretta: i due eventi potrebbero anche essere considerati in parallelo.

Capitolo 3

Soluzione proposta

Il seguente progetto si pone l'obiettivo di trovare un algoritmo che utilizzi un approccio differente e, soprattutto, meno rigido di quello applicato dal Building Instance Graph. L'idea alla base è quella di sviluppare un meccanismo che si ripete in modo iterativo e che sia in grado di proporre delle soluzioni alternative in modo pseudo-casuale. Tali soluzioni verranno poi valutate in una fase successiva e accettate o meno in base a vincoli meno stringenti rispetto a quelli di BIG.

3.1 Iterated Local Search

Il meccanismo iterativo implementato si appoggia alla *Iterated Local Search* [LMS03].

L'idea alla base di questo approccio è la costruzione iterativa di una sequenza di soluzioni ammissibili. Partendo da una soluzione (*soluzione corrente*), si applicano delle mosse migliorative (*fase di perturbazione*) al fine di esplorarne il vicinato e pervenire, sperabilmente, ad una soluzione migliore. La fase di perturbazione è sempre seguita da una *fase di accettazione*, in cui si decide se la nuova soluzione determinata possa sostituire o meno quella corrente. In particolare, se la nuova soluzione é migliore di quella

corrente, questa è accettata e diventa la nuova soluzione da cui far partire la fase di perturbazione. In tal caso, si valuta anche se tale soluzione é migliore della soluzione ottima corrente. Se si, anche questa viene aggiornata.
Qualora la nuova soluzione non sia migliore di quella corrente, al fine di evitare che l'approccio rimanga intrappolata negli ottimi locali, si decide sulla sua accettazione mediante un criterio probabilistico, ereditato dall'approccio di *Simulated Annealing* [EKÜ17]. L'idea alla base di questo criterio probabilistico é che nelle prime iterazioni, si accettino, con piú elevata probabilitá soluzioni peggiorative. Tale probabilitá tende a diminuire mano mano che l'approccio si avvicina al *criterio di terminazione* fissato.

La bontà del risultato che si ottiene dipende dalle perturbazioni applicate: se sono troppo piccole si potrebbe ricadere sempre nella stessa soluzione ottima locale; mentre se sono troppo grandi equivale all'applicazione di una ricerca randomica.

3.2 Funzione obiettivo

Per valutare in termini numerici la bontà delle soluzioni ottenute dall'algoritmo è stato necessario pensare ad un criterio di misura della bontá (*funzione obiettivo*). I criteri di qualità scelti sono i seguenti:

- Generalizzazione dell'IG corrente: $gen(IG(\sigma))$;
- Differenza di archi dell'IG corrente rispetto all'IG estratto inizialmente: $\#move$.

$$fo(IG(\sigma)) = gen(IG(\sigma)) + (\#move * p)$$

Per *generalizzazione* s'intende il valore del numero di tracce che è possibile generare su quel determinato IG. Se il grafo è strettamente sequenziale si avrà $gen(IG(\sigma)) = 1$, che è anche il valore minimo che può assumere.

Nell'esempio in fig 1.2: $gen(IG(\sigma_1)) = 10$: il parallelismo che parte dall'evento *c* e termina in *g* permette di generare 10 diverse sequenze di eventi che è possibile tradurre in 10 diverse tracce.

Quando si parla di differenza di archi s'intende la seguente formula:

$$\#move = \#edge_{remove}(IG(\sigma)) + \#edge_{add}(IG(\sigma)) - (k * 2)$$

Vengono conteggiati sia gli archi che erano presenti nell'IG estratto all'inizio dell'algoritmo ($IG_{Start}(\sigma)$) e che nell'IG corrente sono stati rimossi ($\#edge_{remove}(IG(\sigma))$), sia la situazione duale, ovvero quegli archi che non erano presenti in $IG_{Start}(\sigma)$ ma lo sono in quello corrente ($\#edge_{add}(IG(\sigma))$). Tale somma viene diminuita di $k * 2$ per rendere ammissibili un certo numero di mosse dato dal numero di archi rimossi k e dallo stesso numero di archi che potrebbero essere aggiunti.

Questo dato viene pesato nella funzione obbiettivo per dare la possibilità alla local search di effettuare delle perturbazioni e di allontanare l'IG dalla struttura che aveva inizialmente, ma di non farlo in modo eccessivo. Dando un peso maggiore agli archi aggiunti o rimossi in eccesso si limita questo allontanamento dallo standard: è per questo che nella funzione obiettivo finale tale valore verrà moltiplicato per *p*, per aggiungere una penalizzazione.

3.3 Outline della metaeuristica proposta

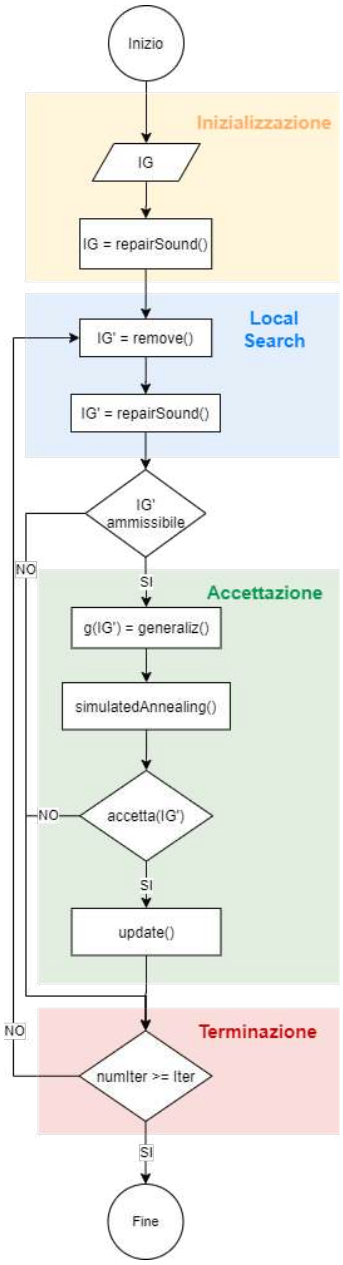

Figura 3.1: Flow chart della metaeuristica

Con l'ausilio della fig. 3.1, prima di passare alla spiegazione dettagliata, s'illustrano brevemente i passaggi della metaeuristica a livello macroscopico, dividendola in:

1. **Inizializzazione** Si estrae l'IG basandosi sulla traccia corrente, non conforme, e sul modello di processo. Utilizzando queste informazioni si ricava un instance graph il più delle volte non *ammissibile*. Per questo motivo è necessario effettuare una prima riparazione andando ad aggiungere degli archi che rendano sound l'IG.

2. **Local Search** Si effettuano le modifiche partendo da un IG ammissibile. Si rimuovono k archi e, per ognuno di questi, ripristinati i percorsi eliminati: se si elimina un arco e si aggiungeranno uno o più archi che permettano di ripristinare il collegamento che e generava. Per concludere questa fase si controlla se l'IG ottenuto è sound, in caso contrario verrà reso tale.

3. **Accettazione** Una volta verificato che IG' sia ammissibile si passa al calcolo della funzione obiettivo. In base all'esito ottenuto dal confronto con le funzioni obiettivo dell'IG corrente e di quello ottimo e, eventualmente, dal simulated annealing si procede all'aggiornamento dell'IG corrente e/o ottimo. Se IG' viene scartato non viene fatto alcun aggiornamento e si passa alla fase successiva.

4. **Terminazione** Terminate le tre fasi precedenti si controlla il numero di iterazioni eseguite fino a quel momento: raggiunto il numero massimo l'algoritmo termina.

3.4 Simulated Annealing

Per poter ovviare al problema di rimanere intrappolati in soluzioni ottime locali, si è deciso di adottare il criterio di accettazione ereditato dall'approccio di *Simulated Annealing*.

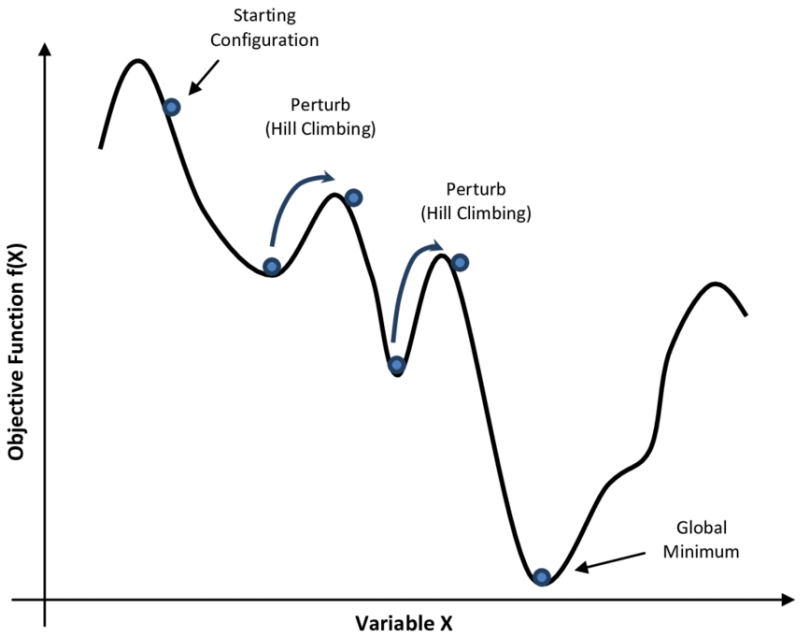

Figura 3.2: Simulated Annealing [GKT16]

La figura 3.2 illustra in modo grafico il funzionamento di questa strategia. Nei punti di minimo locale ci si trova di fronte ad una soluzione localmente ottima dalla quale non si riesce più ad allontanarsi compiendo una sola mossa di perturbazione della local search implementata: spostandosi di poco rispetto al punto che si trova nella "conca" si troveranno solamente soluzioni peggiorative che, di conseguenza, non saranno accettate.

Per poter uscire da questa situazione di stallo è necessario che l'algoritmo accetti delle soluzioni peggiorative rispetto a quella corrente, in modo da poter cercare sperabilmente una soluzione di ottimo globale. Per far sì che la scelta avvenga nel modo corretto è necessario impostare due parametri iniziali:

- **Fattore di deterioramento** $\alpha \in [0, 1]$;
- **Temperatura (T)**: diminuirà ad ogni iterazione per opera del fattore di deterioramento applicando la seguente moltiplicazione

$$T = T * \alpha \tag{3.1}$$

La formula che si applica per scegliere se accettare o meno una soluzione peggiore è riportata nel seguito.
Sia $z \in [0, 1]$ un numero casuale e siano:

$$\delta = \frac{fo(IG(\sigma)) - fo(IG'(\sigma))}{T} \tag{3.2}$$

$$prob = e^{\delta} \tag{3.3}$$

La soluzione è accettata se

$$prob > z \tag{3.4}$$

La variazione di temperatura è definita in modo che:

- Se T è elevata sono ammessi salti più alti: quando si trova un minimo si prova a proseguire ipotizzando che sia un minimo locale;
- Se T è bassa sono ammessi meno salti: la probabilità che venga scelta una soluzione peggiorativa è più bassa;
- Più α si avvicina a 0 più veloce sarà la diminuzione di T ad ogni iterazione fino ad arrivare ad una temperatura tendente a 0 che porta a non accettare più soluzioni peggiorative congelando la soluzione in quel minimo;
- Più α si avvicina a 1 più lenta sarà la diminuzione di T ad ogni iterazione e, di conseguenza, la probabilità che una soluzione venga scelta. Può implicare il non raggiungimento della conclusione del calcolo e quindi non trovare il minimo globale: se la temperatura non diminuisce abbastanza si continueranno a scegliere molte soluzioni peggiorative.

La scelta dei parametri iniziali T e α dipende dalla tipologia del problema. Per trovare i valori adatti è necessario eseguire delle simulazioni.

3.5 Pseudocodice

In questa sezione viene analizzata nel dettaglio l'implementazione della metaeuristica. Per una migliore comprensione si riporta inizialmente l'elenco dei parametri e delle variabili utilizzate nella scrittura dello pseudocodice.

Parametri:

- $T =$ temperatura iniziale
- $\alpha =$ parametro di deterioramento
- $\#iter =$ iterazioni
- $k =$ archi rimossi ad ogni iterazione nella local search

Variabili:

- $cr =$ relazioni causali del modello
- $\sigma = \langle a_1, a_2, ..., a_n \rangle$ traccia non conforme al modello
- $IG_{Best}(\sigma) =$ miglior IG trovato per la traccia σ
- $fo_{Best} =$ funzione obiettivo di $IG(best)$
- $IG(\sigma) =$ ultima soluzione ammissibile accettata
- $fo =$ funzione obiettivo di $IG(\sigma)$
- $IG'(\sigma) =$ soluzione dell'iterazione corrente
- $fo' =$ funzione obiettivo di $IG'(\sigma)$

Si definisce *arco strutturale* ogni arco che si trova nell'IG estratto al primo passaggio dell'algoritmo.

È possibile suddividere in modo modulare lo pseudocodice della metaeuristica. Insieme alla funzione principale, *Main()*, si riportano le seguenti funzioni:

1. ***RepairSound()***: per la riparazione della soundness di un IG;
2. ***Remove()***: corrisponde alla local search, perturba l'ultima soluzione accettata cercando una soluzione migliore;
3. ***Accettazione()***: per il calcolo della funzione obiettivo e la decisione della scelta della nuova soluzione di partenza per l'iterazione successiva.

3.5.1 Main

La funzione *main* è quella principale. Qui si ripropone a grandi linee il flusso di esecuzione che troviamo in fig. 3.1.

Nel flusso di esecuzione c'è una funzione di cui non si è ancora parlato: *Ammissibilità()*. Un IG è *ammissibile* se è sound e se la traccia di partenza fitta in quel grafo, ovvero è possibile percorrere l'IG seguendo nodo per nodo la traccia σ. La funzione andrà ad effettuare un controllo, se queste due caratteristiche sono presenti nell'IG restituisce *True* altrimenti *False*.
È importante effettuare questo controllo perché, come abbiamo detto in precedenza, queste sono le caratteristiche che vogliamo abbia l'IG finale, quindi se così non fosse è inutile proseguire nelle operazioni successive ma si può passare direttamente all'iterazione successiva.

Algorithm 1 $Main()$

$IG(\sigma) = IG_{start}(\sigma) = ExtractInstanceGraph(\sigma, cr)$

$IG(\sigma) = RepairSound(IG(\sigma))$

if σ fitta su $IG(\sigma)$ **then**

{Inizializzazione delle variabili}

$IG(\sigma), fo, IG_{Best}(\sigma), fo_{Best} =$

$= Accettazione(IG(\sigma), 0, 10000, 10000)$

end if

while $\#iter$ **do**

$\#iter- = 1$

$IG'(\sigma) = Remove(IG(\sigma), k)$

$IG'(\sigma) = RepairSound(IG'(\sigma))$

if $Ammissibilita(IG'(\sigma))$ **then**

$\#move =$ archi eliminati/aggiunti rispetto $IG_{start}(\sigma) - k * 2$

if $\#move < 0$ **then**

$\#move = 0$

end if

$T = T * \alpha$

$IG(\sigma), fo, IG_{Best}(\sigma), fo_{Best} =$

$= Accettazione(IG'(\sigma), \#move, fo, fo_{Best})$

end if

end while

3.5.2 Riparare la soundness dell'IG

La funzione *RepairSound()* come suggerito dal nome va a riparare la caratteristica di *soundness* dell'IG corrente.
Viene richiamata nella parte iniziale dell'algoritmo per riparare l'IG di partenza che, con molta probabilità, non sarà sound per colpa della non conformità della traccia considerata e successivamente in ogni local search a seguito delle perturbazioni che sono state effettuate nell'IG corrente. In questo secondo caso è utile andare a riparare il grafo in quanto c'è la possibilità che vengano eliminati degli archi non strutturali e che quindi non venga recuperato il percorso tra quei due eventi lasciando il grafo sconnesso.

L'idea alla base di questa riparazione sta nello scorrere tutti i nodi iniziali e finali in eccesso, che non siano quindi il primo e l'ultimo della traccia, per collegarli ad un altro nodo e connettere il grafo. La ricerca di tale arco parte dal nodo che si vuole collegare e si scorre la traccia indietro (o avanti) fino a trovare una connessione ammissibile secondo le dipendenze delle attività nel modello di processo di riferimento.

Algorithm 2 $RepairSound()$

Require: $IG(\sigma)$

if $IG(\sigma)$ è sound **then**

return $IG(\sigma)$

else

$inodes =$ nodi iniziali escluso il primo

$fnodes =$ nodi finali escluso l'ultimo

end if

for $node$ in $inodes$ **do**

$fnode = node$

while 1 **do**

$inode =$ evento nella traccia che precede $node$

if $(inode, fnode)$ è tra gli archi ammissibili and $inode$ è nelle dipendenze di $fnode$ **then**

$IG'(\sigma) = add((inode, fnode))$

$break$

else

$node = inode$

end if

end while

end for

for $node$ in $fnodes$ **do**

procedimento duale a quello per gli $inodes$

end for

return $IG'(\sigma)$

3.5.3 Local Search

In questo paragrafo verrà mostrato il funzionamento della *Remove()* che unita ad una *RepairSound()* formano la local search della metaeuristica implementata.

Tale funzione è il fulcro della perturbazione che viene effettuata ad ogni iterazione. Partendo da un IG, si elimina un arco e in modo casuale. Tra gli archi da eliminare è possibile scegliere, con la stessa probabilità degli altri, anche di non eliminare alcun arco. Una volta effettuata la scelta, se e è strutturale allora viene ripristinato il percorso che è stato eliminato cercando una via alternativa. Se l'unico modo per collegare quei due eventi è reinserire e stesso, verrà aggiunto nuovamente e sarà stata effettuata una rimozione a vuoto. Il parametro *k* indica il numero di archi che verranno eliminati: il procedimento sopra descritto verrà quindi ripetuto k volte.

Algorithm 3 $Remove()$

Require: $IG(\sigma), k$

  while k **do**

    $k = k - 1$

    $edge =$ arco random da $IG(\sigma)$

    $IG'(\sigma) = remove(edge)$

    if $edge$ è un arco strutturale **then**

      ripristina il percorso eliminato

    end if

  end while

  return $IG'(\sigma)$

3.5.4 Accettazione dell'IG

Nell'*accettazione* si calcola la funzione obiettivo di $IG'(\sigma)$, ovvero l'IG che si sta modificando nell'iterazione corrente, e, se sono soddisfatti i vincoli, aggiorna i valori di $IG(\sigma)$ e $IG_{Best}(\sigma)$.

Per effettuare il calcolo della generalizzazione si utilizza la funzione di una libreria di python (*pm4py*) che simula le possibili tracce su un *process tree* (PT): struttura molto più veloce da analizzare rispetto ad una rete di Petri.
Per ottenere un PT è necessario avere una *rete di Petri* (PN), che a sua volta si ottiene dall'IG. In certe situazioni non è possibile completare questa trasformazione da PN a PT: quando il modello ottenuto non è *block structured* [LFVDA13]. In questi casi la funzione per il calcolo della generalizzazione utilizzerà una funzione basata sulla PN stessa. Questo secondo metodo però è molto più oneroso rispetto al primo e potrebbe portare a dei problemi quando la metaeuristica verrà applicata a dei dataset più complessi.

Ottenuta la generalizzazione da una delle due funzioni di pm4py ci si può trovare in una delle seguenti situazioni:

1. Se la funzione obiettivo calcolata (fo') è uguale a quella corrente (fo) si sceglie in modo randomico se tenere come soluzione corrente quella attuale ($esito = False$) oppure se prendere il nuovo IG ($esito = True$);

2. Se fo' è peggiorativa rispetto a fo s'invoca il Simulated Annealing che deciderà se scartare la soluzione ($esito = False$) o meno ($esito = True$);

3. Se fo' è migliorativa rispetto a fo oppure in una delle due situazioni precedenti è stato dato $esito = True$, fo' viene scelta come nuova soluzione corrente: $fo = fo'$.

    - Se fo' è migliorativa anche rispetto alla miglior funzione obiettivo trovata fino a quel momento (fo_{Best}) allora viene aggiornato anche il miglior IG ($IG_{Best}(\sigma)$) e cancellata la lista degli IG trovati con quella funzione obiettivo;

- Se fo' è uguale a fo_{Best} allora IG' viene aggiunto alla lista dei miglior IG trovati con quella funzione obiettivo.

Algorithm 4 $Accettazione()$

Require: $\#move, IG(\sigma), fo, IG_{Best}(\sigma), fo_{Best}$

$tree = createProcessTree(IG(\sigma))$

$fo' = generalizzazione(tree)$

if $\#move < 0$ **then**

$\#move = 0$

end if

$fo' = fo' + \#move * 5$

if $fo' = fo$ **then**

$esito = random(True, False)$

end if

if $fo' > fo$ **then**

$esito = simulatedAnnealing(fo', fo)$

end if

if $fo' < fo$ *or* $esito = True$ **then**

$fo = fo'$; $IG(\sigma) = IG'(\sigma)$

if $fo' < fo_{Best}$ **then**

$fo_{Best} = fo'$; $IG_{Best}(\sigma) = IG'(\sigma)$; $list_{Best}.clear()$

else if $fo' = fo_{Best}$ **then**

$list_{Best}.append(IG'(\sigma))$

end if

end if

return $IG(\sigma), fo, IG_{Best}(\sigma), fo_{Best}$

Capitolo 4

Sperimentazione

In questo capitolo si riportano i risultati ottenuti dall'esecuzione della metaeuristica.

4.1 Mini dataset

Durante tutta la fase d'implementazione e le prime sperimentazioni è stato utilizzato un dataset molto semplice formato dalle seguenti tracce:

- $\sigma_1 = \langle a, b, d, f, e, x, g, l, n, o \rangle$
- $\sigma_2 = \langle a, b, c, x, f, g, h, d, i, o \rangle$
- $\sigma_3 = \langle b, c, d, e, f, x, g, m, n \rangle$
- $\sigma_4 = \langle a, c, f, d, x, e, l, n, o \rangle$
- $\sigma_5 = \langle a, l, b, c, x, f, d, e, g, m, n, o \rangle$

Ogni traccia ha una caratteristica strutturale diversa che è già testata anche nell'algoritmo di riferimento BIG e dal quale si dovrebbe ottenere un buon risultato.

Facendo riferimento al modello in fig. 1.1:

σ_1 : manca l'attività *c*, si ha un move on log che su BIG corrisponde ad un'attività eliminata. Questa attività si trova all'inizio di un parallelismo;

σ_2 : mancano le attività *d, e, n*. Le prime due corrispondono a delle attività eliminate consecutive nel modello, in particolare così non viene eseguito un intero ramo di un parallelismo. Viene anche inserita l'attività *d* in una posizione sbagliata, si ha un move on model;

σ_3 : mancano le attività *a, o* che corrispondono all'attività iniziale e finale del modello;

σ_4 : mancano le attività *b, g*. La seconda potrebbe risultare problematica dato che è l'attività che chiude un parallelismo formato da 3 rami;

σ_5 : è inserita l'attività *l*, che si trova alla fine del modello, in seconda posizione riproducendo la situazione di un'attività inserita di BIG.

Nonostante gli esempi siano molto semplici già da qui si trovano delle divergenze nei risultati, di seguito vengono riportate due figure che mostrano il risultato della metaeuristica e di BIG per le tracce σ_1 e σ_2.

Ricordando che gli archi strutturali sono quelli appartenenti agli IG estratti (con il medesimo approccio) all'inizio di entrambi gli algoritmi, sono presenti tre tipologie di archi nelle figure:

- Archi strutturali rimasti anche nell'IG finale: frecce continue;
- Archi inseriti dall'algoritmo non presenti tra quelli strutturali: frecce tratteggiate;
- Archi strutturali eliminati dall'algoritmo: linee tratteggiate più sottili.

Traccia σ_1 In questa traccia non conforme al modello è stata eliminata l'attività *c*, inizio del parallelismo composto dalle attività *[d,e,f,x]*.
Dalla fig. 4.1 si può notare una differenza strutturale nel risultato: la metaeuristica utilizza l'evento *d* come sostituta alla mancante vincolando quest'attività ad essere eseguita prima di tutte le altre del parallelismo; BIG, invece, utilizza *b* come sostituto, non ponendo vincoli sulla sequenza di esecuzione di *d*.

Nei termini della funzione obiettivo la metaeuristica ottiene una minor generalizzazione ma, osservando i tratteggi, un maggior numero di archi che si differenziano dall'IG iniziale.

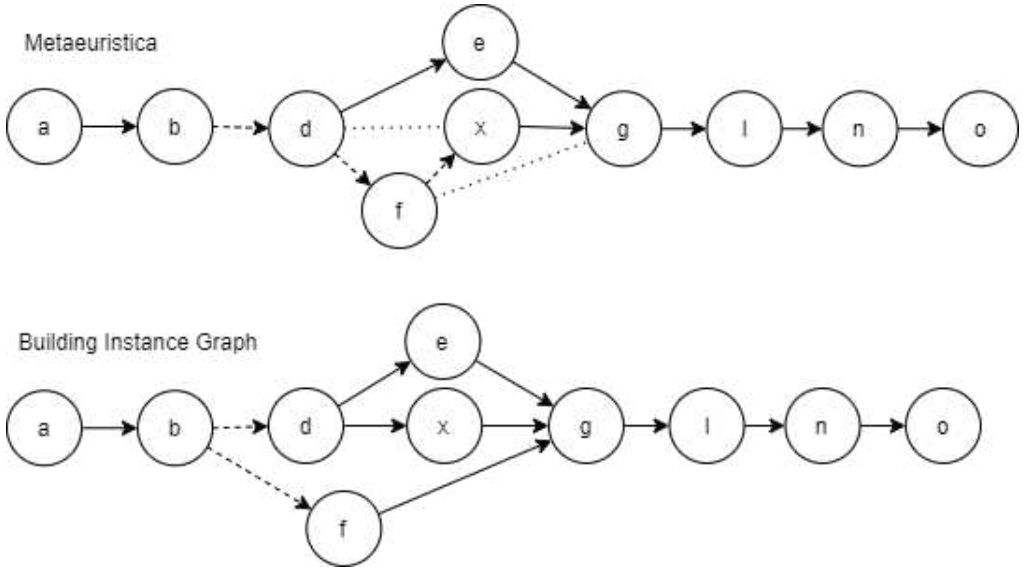

Figura 4.1: Risultati della traccia σ_1

Traccia σ_2 In questo secondo esempio sono presenti due attività eliminate consecutive (*d, e*). Si nota in fig. 4.2 come la metaeuristica sia arrivata ad un risultato ammissibile, al contrario dell'algoritmo preesistente: BIG restituisce un IG non sound nel quale gli eventi *a, x* sono entrambi nodi iniziali.

Per quanto riguarda, invece, l'attività inserita *d* si comportano nello stesso modo. Questa riparazione è corretta dato che quell'evento deve essere inserito in modo sequenziale nella posizione in cui si trova anche se, da modello, non dovrebbe esserci.

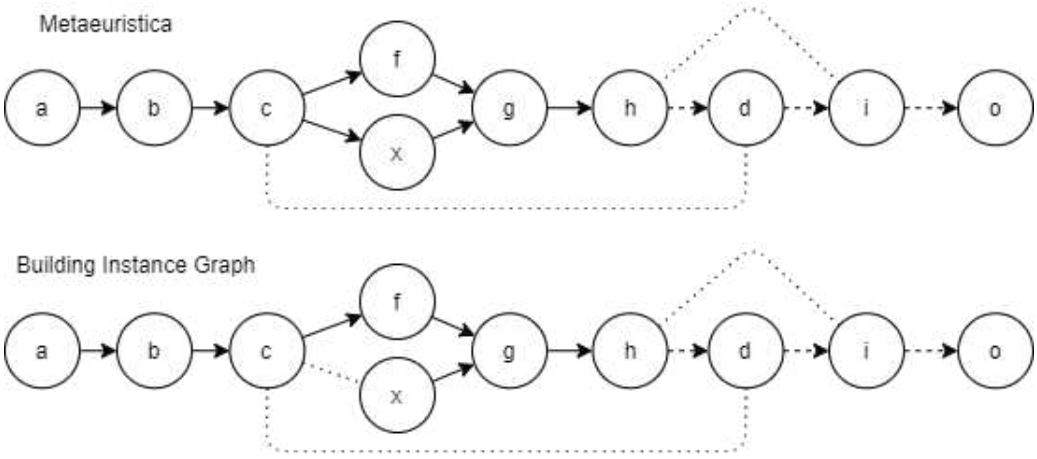

Figura 4.2: Risultati della traccia σ_2

Per quanto riguarda le tracce successive i due algoritmi si comportano in modo molto simile tra loro: non ci sono problemi di riparazione nè su σ_3 quando mancano le attività iniziale e finale, nè su σ_4 per la chiusura del parallelismo.

Questi due esempi sono serviti per una prima panoramica delle soluzioni che si ottengono dai due algoritmi.

4.2 Esempio esteso

4.2.1 Dataset

Per il lavoro di sperimentazione si è utilizzato un modello ed un event log differenti da quelli utilizzati fino a questo punto del documento.

Si è passati da un modello semplice ad uno più complesso nel quale sono presenti anche transizioni ”fantasma”, dette τ; sono identificate in fig. 4.3 dai quadrati neri.

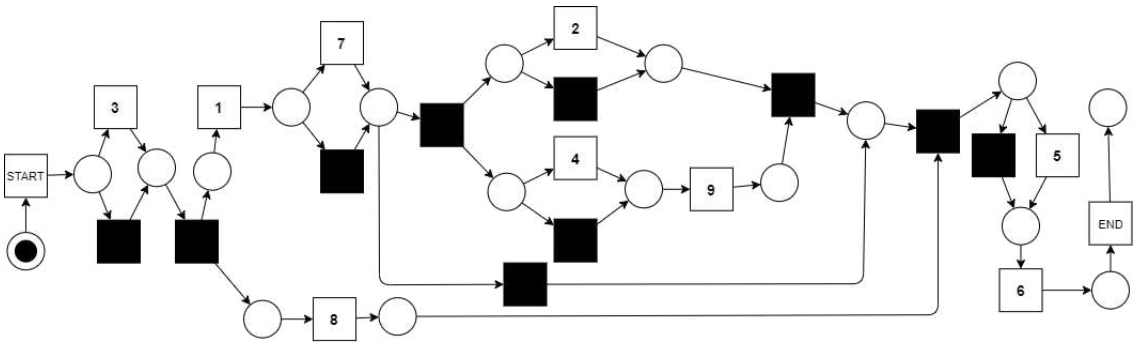

Figura 4.3: Modello di processo del dataset sperimentale

Avere delle τ nel modello permette di ottenere delle tracce dove le attività devono essere eseguite in un certo ordine ma senza essere obbligatorie. Di conseguenza sono presenti anche relazioni causali tra attività che, a colpo d'occhio, non sembra siano consecutive: $(1, 9)$ e $(1, 6)$ sono due esempi di relazioni causali che non sarebbero esistite se le τ fossero state delle attività vere e proprie.

L'event log del dataset, e di conseguenza le esecuzioni degli algoritmi, faranno riferimento a questo modello. Le tracce del dataset sono più di 3000 ma, quelle effettivamente analizzate, sono solo 142.

Ci si riduce ad un numero limitato di tracce in quanto sono state filtrate, ovviamente, quelle conformi e, in seconda battuta, è stata fatta una fase di preprocessing dove le tracce che presentano lo stesso ordine di attività, vengono considerate una sola volta.

Le tracce sono in realtà univoche dato che gli eventi saranno sicuramente eseguiti in tempi e, forse da risorse, differenti. Tuttavia BIG e la metaeuristica non considerano questi dettagli e si basano solo sulla sequenza di esecuzione. La fase di preprocessing serve appunto per applicare l'algoritmo in sequenze di eventi univoche.

4.2.2 Scelta dei parametri

Come anticipato nel capitolo 3, è necessario settare dei parametri prima di lanciare l'esecuzione. La loro scelta è molto importante ed un certo settaggio può portare ad una soluzione migliore rispetto ad un altro. Per questo sono state fatti diversi tentativi variandoli nei seguenti range:

- Temperatura: $T \in (500, 300)$
- Fattore di deterioramento: $\alpha \in (0.7, 0.8, 0.9)$
- Archi rimossi per local search: $k \in (1, 2, 3)$
- $iterazioni \in (100, 300, 500)$
- Peso degli archi: $p \in (5, 10, 20)$

Non esiste un set di parametri migliore in assoluto ma è necessario valutare in base alle caratteristiche del dataset utilizzato. Analizzando i risultati ottenuti si è scelto di utilizzare i seguenti valori:

- Per il simulated annealing: $T = 500$ e $\alpha = 0.8$
  Fanno sì che vengano accettati un buon numero di risultati peggiorativi.
- Per la local search: $k = 2$
  Le tracce hanno in media una lunghezza di 7 eventi che, mappati sull'IG, diventeranno 7 nodi. In generale non si hanno degli IG molto grandi e le perturbazioni possono essere limitate a 2 archi per ogni local search;

- $iterazioni = 100$

  Per lo stesso motivo del punto precedente, le combinazioni di archi da rimuovere è limitato e per arrivare ad una soluzione ottima 100 iterazioni sono abbastanza;

- Penalizzazione degli archi in eccesso nella funzione obiettivo: $p = 5$

  Gli IG con una media di 7 nodi hanno dei parallelismi con un numero di eventi limitati, di conseguenza i valori della generalizzazione rimangono relativamente bassi. Se il peso degli archi in eccesso fosse più alto, la funzione obiettivo dipenderebbe dalla differenza degli archi e non più dal valore calcolato della generalizzazione.

Nell'esecuzione con questo settaggio il simulated annealing ha accettato in media $4,5$ soluzioni peggiorative con un minimo di 0 ed un massimo di 9 per ogni traccia.
Si è notato però che, con tracce di questo tipo, semplici e non troppo elaborate, l'algoritmo trova già nelle prime iterazioni la soluzione migliore che poi verrà restituita alla sua terminazione.

Dopo poco meno di 10 minuti di esecuzione l'algoritmo ha restituito gli IG riparati delle 142 tracce del dataset.

4.2.3 Algoritmi a confronto

In questa sezione l'esecuzione dell'algoritmo della metaeuristica è stata affiancata da quella del Building Instance Graph per poter riportare un confronto nella bontà dei risultati dell'uno e dell'altro nei termini della funzione obiettivo utilizzata in questo documento.

Un primo importante risultato a favore della metaeuristica è la rivelazione che riesce ad ottenere un risultato valido anche per diverse tracce in cui BIG estrae un IG non sound (come nel caso dell'esempio in fig. 4.2). Il 10% ca. delle 142 tracce non ha un risultato valido se viene applicato BIG, al contrario della metaeuristica che non ottiene un risultato solo per una traccia.

Senza considerare le 15 tracce che non hanno portato ad un risultato da parte di BIG, si è effettuato un confronto in termini della funzione obiettivo valutando migliore l'algoritmo che ha ottenuto un IG con generalizzazione minore e, in casi di uguaglianza, quello che ha riportato un numero di archi che si discostano da quelli strutturali minore. Nel caso in cui entrambi i valori sono uguali si decreta un pareggio.
BIG risulta essere migliore in 42 tracce, contro le 57 della metaeuristica, 27 volte invece si arriva ad una situazione di pareggio.

4.2.4 IG a confronto

Nella simulazione con il mini dataset sono state eseguite riparazioni su casi semplici di tracce non conformi. Si riportano in questa sezione gli IG ottenuti alla fine della riparazione effettuata dai due algoritmi a partire da delle tracce non conformi con delle anomalie che possono risultare più problematiche:

- $\sigma_1 = \langle START, 1, 8, 6, 8, 9, 8, 6, END \rangle$
- $\sigma_2 = \langle START, 1, 8, 8, 8, 9, 9, 8, 6, END \rangle$
- $\sigma_3 = \langle START, 1, 1, 8, 9, 8, 6, END \rangle$

Le tracce sono formate anche da attività ripetute due o più volte. In questa situazione non è più possibile mappare gli eventi nei grafi basandosi sulle etichette, ma è necessario utilizzare un identificatore univoco.
Nelle immagini sottostanti un nodo è rappresentato da una coppia di valori *(identificatore - label)* dove il primo risulta crescente in base all'ordine posizionale dell'evento all'interno della traccia stessa, il secondo è riferito alla label dell'attività eseguita.

Traccia σ_1 La prima traccia è lunga nove eventi e presenta tre *attività inserite* secondo l'algoritmo BIG, di cui 2 consecutive: $(4-6), (5-8), (7-8)$. Nell'immagine in fig. 4.4 si riportano gli IG estratti con BIG e con la metaeuristica, rispettivamente sopra e sotto:

BIG ha estratto una struttura con molti eventi in parallelo che permettono di generare 19 tracce diverse ($gen_{big} = 19$), con una differenza dall'IG strutturale di 8 archi ($move_{big} = 8$). Facendo riferimento al modello di processo (fig. 4.3) si possono notare delle incongruenze non giustificabili: né l'attività 6, né la 9 dovrebbero precedere la 8; l'attività 1 dovrebbe essere eseguita prima di tutte le altre e la 6 alla fine ma in questo IG è possibile generare tracce che non rispettano questi vincoli determinati dal modello.

Metaeuristica ha invece estratto un IG più sequenziale che rispetta l'ordine dello svolgimento degli eventi ed i vincoli descritti nel punto precedente lasciando il parallelismo tra le attività 1 e 8. La generalizzazione è molto più bassa e la differenza degli archi da quelli strutturali aumenta solo di uno rispetto all'altro risultato: $gen_{meta} = 2$, $move_{meta} = 9$.

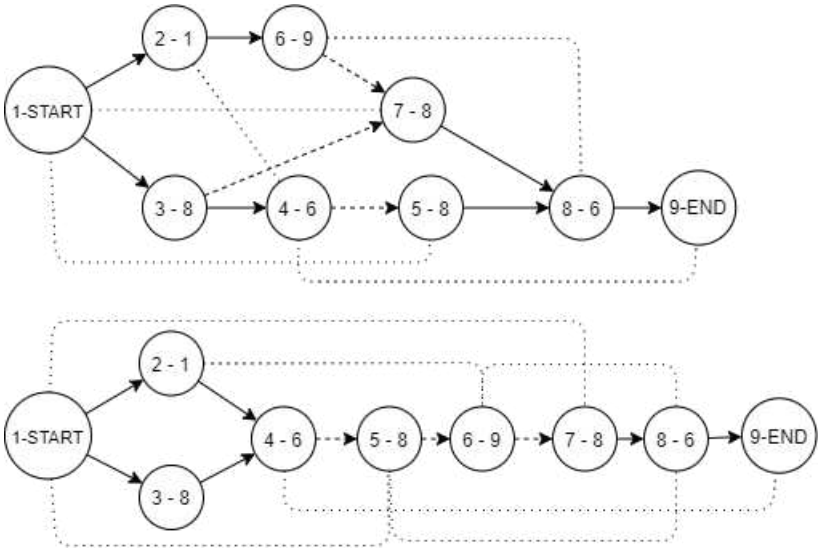

Figura 4.4: Risultati della traccia σ_1

Traccia σ_2 La seconda traccia è lunga dieci eventi e presenta quattro *attività inserite* secondo l'algoritmo BIG: $(4-8), (5-8), (7-9), (8-8)$.
In questo caso, contrariamente al primo in cui la metaeuristica lavora in modo migliore, i risultati possono essere paragonabili e visti come due soluzioni alternative, anche se il secondo ha una generalizzazione migliore secondo i termini della funzione obiettivo. Nell'immagine in fig. 4.5 si riportano gli IG estratti:

BIG : $gen_{big} = 21, move_{big} = 14$.

Metaeuristica : $gen_{meta} = 7, move_{meta} = 14$.

La generalizzazione è molto più bassa in quanto l'attività $(5-8)$ è stata inserita nel ramo superiore del parallelismo invece che in quello inferiore.

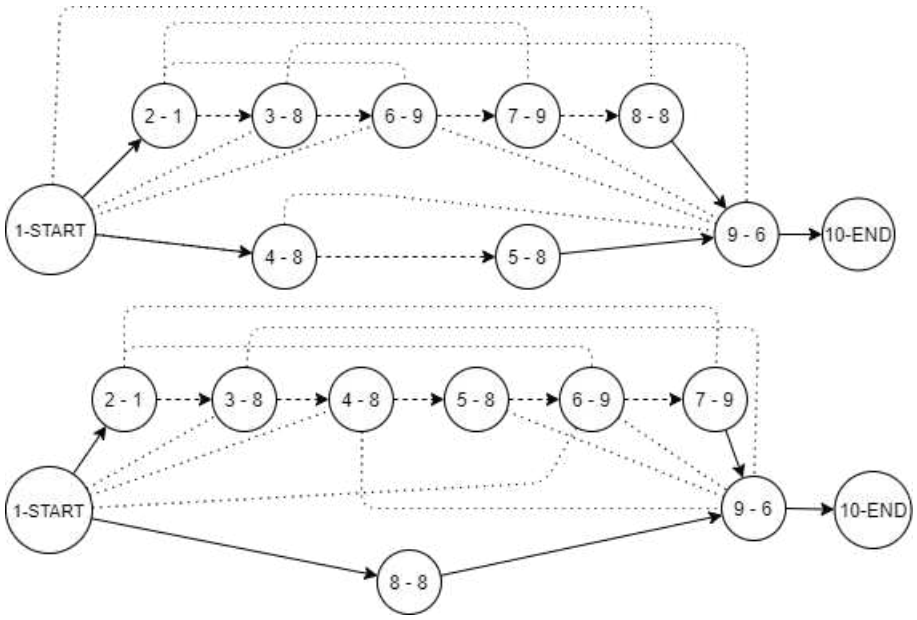

Figura 4.5: Risultati della traccia σ_2

Traccia σ_3 Tuttavia ci sono dei casi in cui la metaeuristica non funziona nel modo corretto e la soluzione ottenuta dal Building Instance Graph è migliore rispetto alla metaeuristica.

BIG in questo caso è riuscito ad ottenere una traccia con una generalizzazione minore ($gen_{big} = 4$, $gen_{meta} = 6$), anche se si è allontanato dall'IG iniziale molto di più rispetto alla metaeuristica ($move_{big} = 10$, $move_{meta} = 4$).
Anche a livello di posizionamento delle attività negli IG, facendo riferimento al modello, BIG ha ottenuto una riparazione migliore rispetto alla metaeuristica soprattutto per quanto riguarda il posizionamento dell'attività 9 che non presenta alcuna relazione causale con l'attività 8.

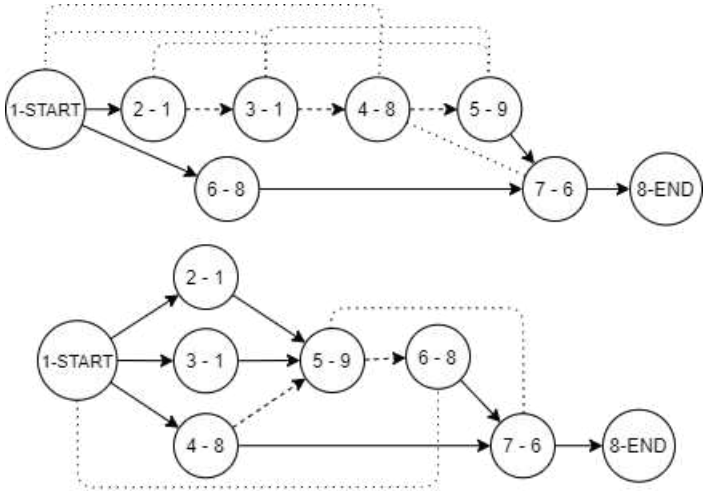

Figura 4.6: Risultati della traccia σ_3

Tuttavia, una fase di sperimentazione più approfondita attraverso l'analisi del settaggio dei parametri, potrebbe portare a soluzioni ancora migliori di quelle ottenute in questo documento.

Capitolo 5

Conclusioni

Nella sezione conclusiva si riportano le limitazioni dell'algoritmo implementato puntando il focus su diversi aspetti che possono tuttavia essere ancora migliorati in sviluppi futuri del progetto di ricerca di cui fa parte.

In termini di tempi di esecuzione è facilmente intuibile che BIG è molto più rapido dovendo applicare delle regole specifiche per la riparazione, mentre la metaeuristica impiega molte più risorse computazionali per svolgere tutte le iterazioni e, di conseguenza, i tempi si dilatano.

Un'altra causa di tale aumento è dovuta dalla fase di accettazione nel momento in cui viene calcolata la generalizzazione dell'IG corrente. Per trovare tale valore è infatti necessario svolgere una funzione di simulazione che generi tutte le tracce possibili per quel grafo.

Come anticipato nella sezione 3.5.4, per calcolare la generalizzazione si utilizza una funzione basata sui process tree, ma non sempre si ottiene dalla local search una rete di Petri block structured. In quest'ultimo caso l'algoritmo utilizza una funzione di simulazione basata proprio su questa rete, un calcolo molto più oneroso e che, nei casi di grafi più complessi, non riesce ad ottenere un risultato per overflow di memoria. In uno sviluppo

futuro dell'algoritmo sarà necessario trovare una differente soluzione per il calcolo di tale metrica.

I tempi della metaeuristica non si dilatano solo nella fase di esecuzione vera e propria, è anche necessaria un'analisi del dataset per poter settare nel modo migliore i parametri. Tuttavia un'attenta analisi iniziale potrebbe non bastare per trovare la combinazione migliore, solo eseguendo più volte l'algoritmo e valutando i risultati è possibile rendersi conto se si sono scelti dei valori relativamente buoni o meno. Non si avrà mai la certezza di aver trovato la soluzione migliore in termini assoluti.

La flessibilità delle regole implementate permette di trovare diverse soluzioni per una stessa traccia. Nello pseudocodice della funzione *Accettazione()* [4] è indicato che la soluzione ottima in realtà potrebbe non essere unica.
Ogni volta che la local search trova un risultato con lo stesso valore (minimo) della funzione di accettazione, questo viene aggiunto ad una lista e non effettuata una scelta se tenere solo l'uno o l'altro. In questo documento tali soluzioni aggiuntive non sono state considerate ma sono un valore aggiuntivo al risultato restituito che permette di valutare IG diversi che abbiano la stessa bontà delle caratteristiche di valutazione.

In conclusione è possibile dire che l'approccio utilizzato per l'implementazione dell'algoritmo ha permesso di ottenere dei buoni risultati riportando alcune limitazioni che potrebbero essere superate in sviluppi futuri di questo progetto.

Bibliografia

[DGPvdA16] Claudia Diamantini, Laura Genga, Domenico Potena, and Wil van der Aalst. Building instance graphs for highly variable processes. *Expert Systems with Applications*, 59:101–118, 2016.

[EKÜ17] Yavuz Eren, İbrahim B Küçükdemiral, and İlker Üstoğlu. Introduction to optimization. In *Optimization in renewable energy systems*, pages 27–74. Elsevier, 2017.

[FvDA15] Dirk Fahland and Wil MP van Der Aalst. Model repair—aligning process models to reality. *Information Systems*, 47:220–243, 2015.

[GKT16] Omid Ghasemalizadeh, Meysam Khaleghian, and Saied Taheri. A review of optimization techniques in artificial networks. *International Journal of Advanced Research*, 4:1668–1686, 09 2016.

[LFVDA13] Sander JJ Leemans, Dirk Fahland, and Wil MP Van Der Aalst. Discovering block-structured process models from event logs-a constructive approach. In *International conference on applications and theory of Petri nets and concurrency*, pages 311–329. Springer, 2013.

[LMS03] Helena R Lourenço, Olivier C Martin, and Thomas Stützle. Iterated local search. In *Handbook of metaheuristics*, pages 320–353. Springer, 2003.

[vdA16] Wil van der Aalst. *Process Mining, Data Science in Action, Second Edition.* Springer, 2016.