Caso di studio in Artificial Intelligence

1 Studenti

Nome Cognome, matricola, mail
Federico Canistro, 775605, f.canistro@studenti.uniba.it
Ivan De Cosmis, 787066, i.decosmis@studenti.uniba.it

2 Introduzione

Con questa relazione presentiamo il progetto di Artificial Intelligence.
Con la prima sezione presentiamo uno script che legge un file CSV contenente dati di eventi e lo converte in un log di eventi. Quindi applica tre diversi algoritmi di scoperta dei processi (Heuristics Miner, Alpha Miner e Inductive Miner) al log degli eventi per scoprire modelli di reti di Petri e successivamente confrontate alcune proprietá. Le reti di Petri scoperte vengono quindi esportate in formato PNML e salvate come immagini.

Con la seconda sezione presentiamo un programma in prolog che legge in input una serie di fatti che definiscono una rete di petri, e converte le informazioni lette in un file in formato **pnml**, uno degli standard piú usati per la rappresentazione delle reti di petri.

Con la terza sezione presentiamo un programma in prolog che legge in input un file **wf** che contiene informazioni che riguardano le transizioni analizzate dal sistema, e restituisce in output una serie di file utili a visualizzare le diverse transizioni.

3 Confronto con sperimentazioni precedenti

Nel caso di studio precedente, si é utilizzata la colonna *timestamp* nel file CSV come identificatore del caso durante la conversione del data frame in un event log. Ció significa che ogni riga nel file CSV veniva trattata come un caso separato con un singolo evento. Di conseguenza, la durata di ogni caso é sempre 0.0 perché é presente solo un evento in ogni caso.
Per calcolare correttamente la durata del caso, é stato necessario specificare una colonna nel file CSV che identifica univocamente ogni caso quando si chiama la funzione *pm4py.format_dataframe*.
Nella nuova versione del codice, stiamo utilizzando la colonna *id_exec* del file CSV come identificatore del caso quando convertiamo il data frame in un event log. Ció significa che le righe nel file CSV che hanno lo stesso valore per la colonna *id_exec*, verranno raggruppate nella stessa traccia dentro l' event log.
La differenza rispetto all' event log precedente é che ora le tracce nell'event log possono contenere piú di un evento. Ció consente di calcolare statistiche come la durata del caso, che misura il tempo trascorso tra il primo e l'ultimo evento in ogni traccia.
Ogni output degli algoritmi utilizzati viene salvato in un file pnml e anche in formato png per poterne osservare la sua visualizzazione grafica.

3.1 Funzioni secondarie

get_all_duration_case: controlla la durata di tutti i case.

compare_pnml_files: Confronta il numero di 'place' e 'transition' nei file.

3.2 Euristic Miner

L'Heuristics Miner é un algoritmo di scoperta di processi piú avanzato che puó gestire meglio le caratteristiche comuni degli event log. Di conseguenza, l'Heuristics Miner é stato in grado di produrre un modello di rete di Petri piú accurato rispetto gli altri algoritmi per lo stesso event log.

3.3 Alpha Miner

L'algoritmo Alpha Miner é un algoritmo di scoperta di processi che puó essere utilizzato per scoprire un modello di rete di Petri da un event log. Tuttavia, l'Alpha Miner ha alcune limitazioni e non é stato in grado di gestire correttamente alcune caratteristiche comuni degli event log, come le attivitá in parallelo e le dipendenze a lungo termine tra le attivitá. In questo caso, l'Alpha Miner ha prodotto un modello di rete di Petri che non rappresenta accuratamente il processo sottostante.

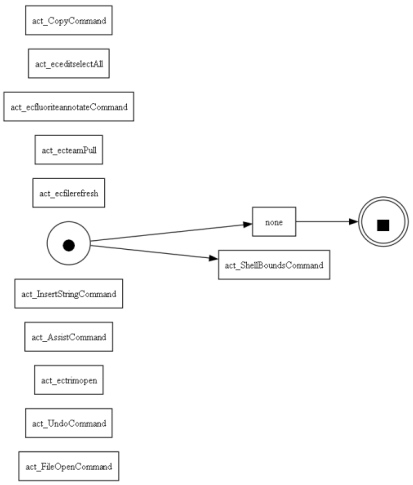

Fig.1. Parte di output alpha sul dataset completo.

3.4 Inductive Miner

L'Inductive Miner é un algoritmo di scoperta di processi che puó essere utilizzato per scoprire un modello di rete di Petri da un event log. In genere, l'Inductive Miner é in grado di gestire event log di grandi dimensioni e complessi in modo efficiente. Tuttavia, in alcuni casi, l'Inductive Miner potrebbe richiedere piú tempo per scoprire un modello di rete di Petri, a seconda delle caratteristiche dell'event log e dei parametri utilizzati.

Nel nostro caso l'inductive miner é stato pi1ú di un'ora in esecuzione sul dataset completo e non si é comunque fermato. Mentre testandolo sui dataset separati ha risposto molto bene.

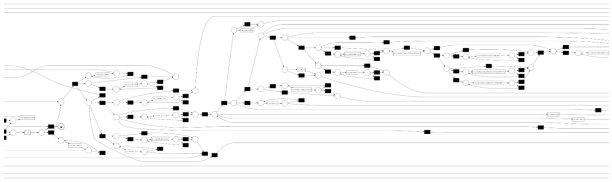

Fig.1. parte di output inductive corretto su uno dei dataset.

Abbiamo inoltre provato ad utilizzare varianti dell'algoritmo Inductive miner, ma sempre senza successo (sempre sul dataset completo). Inoltre, abbiamo anche provato a modificare i parametri dell'algoritmo per migliorarne le prestazioni. Abbiamo utilizzato il parametro *noise_threshold* per specificare una soglia di rumore per l'algoritmo. L'aumento della soglia di rumore puó aiutare l'algoritmo a gestire meglio gli event log rumorosi e migliorare le prestazioni, ma nel nostro caso ció ha portato ad output scorretti come nel caso dell'alpha miner.

4 Conversione rete di Petri da formato Woman a formato pnml

L'idea di questo algoritmo é leggere da un file pl in input tutti i place, transition e arc nel formato:

1. place(X).
2. transition(X).
3. arc(X, Y, T).

Vengono salvati all'interno di tre liste indipendenti (conversion_to_pnml.pl, 1):

```
% Dichiaro dinamiche le liste per
% poter essere modificate a runtime
:- dynamic string_list1/1.
:- dynamic string_list2/1.
:- dynamic string_list3/1.
```

In seguito vengono creati i predicati per accedere alle liste (conversion_to_pnml.pl, 6):

```
% Aggiunge stringa nella lista in input (1,2 o 3)
add_string(ListNumber, String) :-
    (   retract(string_list(ListNumber, List))
    ->  assertz(string_list(ListNumber, [String|List]))
    ;   assertz(string_list(ListNumber, [String]))
    ).

% Restituisce la lista in base al numero in input (1,2 o 3)
get_strings(ListNumber, Strings) :-
    (   string_list(ListNumber, Strings)
    ->  true
    ;   Strings = []
    ).
```

Il predicato **add_string** aggiunge l'elemento in input alla lista di indice in input Il predicato **get_strings** ritorna la lista di indice in input

4.0.1 Contatore

Durante la creazione del file **pnml** é necessario assegnare degli id ai place, transition e arc. Per fare questo usiamo un contatore che viene incrementato ogni volta per ottenere un nuovo id. Codice per gestire il contatore(conversion_to_pnml.pl, 20):

```
% Inizializzo contatore usato nel salvataggio dei dati nel file pnml
:- dynamic(counter/1).
counter(0).

% Inizializzo contatore per gli id dei place, delle transition e degli arc nel file pnml
increment_counter :-
    retract(counter(Count)),
    CountPlusOne is Count + 1,
    assert(counter(CountPlusOne)).
```

4.1 Lettura dei dati dal file

4.1.1 process_file(FileName)

La procedura che legge il file e salva i place, le transition e gli arc all'interno delle liste si chiama **process_file(FileName)**.

Questa procedura riceve in input il percorso del file da leggere (conversion_to_petri.pl, 30).

```
% Processa il file in input ed esporta il file pnml nuovo
process_file(FileName) :-
    retractall(string_list(1, List)),
    retractall(string_list(2, List)),
    retractall(string_list(3, List)),
    consult(FileName),
    process_net,        % Inizia ad analizzare la rete
    get_strings(1, Places),
    get_strings(2, Transitions),
    get_strings(3, Arcs),
    ask_folder_path(Path),
    write_pnml(Path, Places, Transitions, Arcs).
```

"retractall" viene usato per azzerare tutte le liste ogni volta che viene consultato il file. "process_net" é la procedura che processa la rete.

4.1.2 process_net

In questa procedura troviamo le operazioni che vengono eseguite per estrarre i dati dai fatti letti in input:

1. extract_places(NewPlaces): estrae tutti i place dal database e li salva all'interno della lista NewPlaces
2. copy_list(1, NewPlaces): copia la lista in input nella lista dinamica 1
3. extract_transitions(NewTransitions): estrae tutti le transition dal database e li salva all'interno della lista NewTransitions
4. copy_list(2, NewTransitions): copia la lista in input nella lista dinamica 2
5. extract_arcs(NewArcs): estrae tutti gli arc dal database e li salva all'interno della lista NewArcs
6. copy_list(3, NewArcs): copia la lista in input nella lista dinamica 3

(conversion_to_pnml.pl, 69)

```
% Estraggo le informazioni dai fatti
process_net :-
    extract_places(NewPlaces),
    copy_list(1, NewPlaces),
    extract_transitions(NewTransitions),
    copy_list(2, NewTransitions),
    extract_arcs(NewArcs),
    copy_list(3, NewArcs).
```

4.1.3 extract_places(Places)

Analizziamo questa procedura che é molto simile a **extract_transitions(Transitions)** e **extract_arcs(Arcs)**, cambia solamente la sintassi in base al tipo di dato da estrarre (conversion_to_pnml.pl, 49):

```
% Estraggo tutti i place in una lista
extract_places(Places) :-
    findall(Place, place(Place), Places).
```

Findall ricerca tuttii fatti nel formato **place(Place)** e li salva nella lista **Places**.

4.1.4 copy_list

Con questo predicato abbiamo due casi (conversion_to_pnml.pl, 61):

1. copy_list(IdList, []): Caso in cui la lista in input é vuota e quindi abbiamo finito la copia
2. copy_list(IdList, [H|T]): Caso in cui la lista bisogna ancora copiarla tutta

```
% Predicato che ci permette di copiare una lista in una delle 3 liste
% dinamiche dato indice in input
copy_list(IdList, []) :-
    true.

copy_list(IdList, [H | T]) :-
    add_string(IdList, H),
    copy_list(IdList, T).
```

Con questa funzione abbiamo finito di analizzare la parte relativa alla lettura dei dati da file in input.

4.2 Scrittura del file pnml

Per scrivere nel file pnml utilizziamo la procedura **write_pnml**.

4.2.1 write_pnml

Questa funzione apre il file del percorso inserito in console e inizia a scrivere le varie parti del file pnml (conversion_to_petri.pl, 78).

```
% Scrivo il file pnml dati place, transition e arc
write_pnml(File, Places, Transitions, Arcs) :-
    open(File, write, StreamWrite),
    write_header(StreamWrite),
    write_places(StreamWrite, Places),
    write_transitions(StreamWrite, Transitions),
    write_arcs(StreamWrite, Arcs),
    write_footer(StreamWrite),
    close(StreamWrite).
```

4.2.2 ask_folder_path(String)

Con questa procedura chiediamo il percoso in cui salvare il file di output. Al percorso inserito viene attaccata la stringa che contiene il nome del file di output attraverso **atom_concat**(conversion_to_petri.pl, 43).

```
ask_folder_path(String) :-
    write('Enter path where to save output file(e.g. C:/User/Desktop): '),
    read_line_to_string(user_input, Read),
    atom_concat(Read, "/petri_net.pnml", String).
```

4.2.3 write_header(StreamWrite)

Questa funzione scrive l'header del file pnml (conversion_to_petri.pl, 88).

```
% Scrivo intestazione file
write_header(StreamWrite) :-
    write(StreamWrite, '<?xml version="1.0" encoding="UTF-8"?>'), nl(StreamWrite),
    write(StreamWrite, '<pnml xmlns="http://www.pnml.org/version-2009/grammar/pnml">'), nl(StreamWrite),
    write(StreamWrite, '  <net id="net1" type="http://www.pnml.org/version-2009/grammar/ptnet">'), nl(StreamWrite),
    write(StreamWrite, '    <name>'), nl(StreamWrite),
    write(StreamWrite, '        <text> Petri Net </text>'), nl(StreamWrite),
    write(StreamWrite, '    </name>'), nl(StreamWrite),
    write(StreamWrite, '    <page id="page1">'), nl(StreamWrite).
```

4.2.4 write_places, write_transitions, write_arcs

Dato che sono tutti molto simili, cambia solamente la formattazione per rispettare il formato pnml, prendiamo in analisi **write_places** Questa procedura ha due casi:

1. Caso base (conversion_to_petri.pl, 99):

   ```
   % Questo predicato serve quando abbiamo finito tutti i place
   write_places(StreamWrite, []) :-
       true.
   ```

   Questo é il caso in cui sono finiti i place da scrivere su file

2. Caso con lista ancora da svuotare (conversion_to_petri.pl, 103):

   ```
   % Questo predicato serve quando ci sono ancora place da scrivere
   write_places(StreamWrite, [Place|Places]) :-
       write(StreamWrite, '        <place id="'),
       counter(Count),
       write(StreamWrite, Count), write(StreamWrite, '">'), nl(StreamWrite),
       increment_counter,
       write(StreamWrite, '            <name>'),
       nl(StreamWrite),
       write(StreamWrite, '                <text>'),
       write(StreamWrite, Place), write(StreamWrite, '</text>'), nl(StreamWrite),
       write(StreamWrite, '            </name>'), nl(StreamWrite),
       write(StreamWrite, '        </place>'), nl(StreamWrite),
       write_places(StreamWrite, Places).
   ```

   In questo caso ci sono place da scrivere su file e grazie alla funzione write andiamo a scrivere il place rispettando la sintassi del formato pnml. "increment_counter" aumenta il contatore che rappresenta l'id dei place, transition e arc. Infine la funzione richiama se stessa iterando con la lista senza il primo elemento.

4.2.5 write_footer

Questo predicato scrive il pié di pagina del file pnml (conversion_to_petri.pl, 159)

```
% Questo predicato serve quando dobbiamo scrivere il pie di pagina pnml
write_footer(StreamWrite) :-
    write(StreamWrite, '    </page>'), nl(StreamWrite),
    write(StreamWrite, '  </net>'), nl(StreamWrite),
    write(StreamWrite, '</pnml>'), nl(StreamWrite).
```

Con questa procedura abbiamo finito di descrivere il codice di questa sezione

4.2.6 Descrizione del caso di test

Questo programma viene caricato in SWI-Prolog usando il comando:
consult('Percorso_fino_al_file/conversion_to_petri.pl').
In seguito viene richiamata la procedura:
process_file('Percorso_fino_al_file/petri_net.pl').
Alla richiesta del percorso dove salvare il file inserire:
Percorso_fino_a_dove_salvare_il_file/nome_file.pnml

Il programma restituirá true e salverá nel percorso indicato il file risutalto. Nel nostro caso nella repository github il file é:
conversione_petriWoman_pnml/petri_net.pnml

5 Visualizzazioni delle transizioni da un file wf

In questa sezione andiamo a scrivere all'interno di un file di testo le possibili transizioni successive, in base all'azione di input e alla corrispondenza tra casi. Inoltre vengono generati due file dot, aventi diverse proprietá, che sono utilizzati per generare graficamente il grafo che rappresenta la continuitá tra transizioni. Inoltre vengono anche indicate delle possibili transizioni, che anche se non sono mai state osservate dai casi del sistema, potrebbero comunque avvenire.

5.0.1 Gestione del modulo

Per poter utilizzare le transition caricate abbiamo bisogno di capire il modulo in cui appartengono. Per evitare di fare hardcode con il nome del modulo usato per i test (monday) e quindi forzare sintassi del tipo:
monday:transition(Input-Output,Cases,Repetition)
Abbiamo utilizzato una variabile dinamica in cui salviamo l'atomo del modulo letto da tastiera(model_explorator.pl, 1).

```
% Soluzione per poter rendere meno vincolante il codice al modulo
% inserito.
% In questo modo basta inserire il nome del modulo da cui sono state
% lette le transition per poter visualizzare nel file i dati
% senza la necessita di fare hardcode col nome del modello
:- dynamic module/1.

% Predicato che imposta il modulo
set_module(Atom) :-
    retractall(module(_)),
    assertz(module(Atom)).

% Predicato che legge il modulo
get_module(Atom) :-
    module(Atom).
```

Il modulo viene chiesto dalla procedura **ask_module_name(String)** (model_explorator.pl, 164):

```
% Chiedo il nome del modulo usato
ask_module_name(String) :-
    write('Enter the module name used: '),
    read_line_to_string(user_input, String).
```

5.0.2 Gestione della transizione in analisi

Durante l'esecuzione del codice é necessario recuperare le informazioni della transizione in analisi durante la stampa delle transizioni sui diversi file, per questo abbiamo usato un'altra variabile dinamica capace di memorizzare la transizione in analisi (model_explorator.pl, 27)

```
% Variabile dinamica che viene utilizzata per salvare la transizione in analisi e poter
% recuperare le sue informazioni da qualsiasi parte del codice
:- dynamic current_transition/1.

% Predicato per impostare la transizione in analisi
set_current_transition(Transition) :-
    retractall(current_transition(_)),
    assertz(current_transition(Transition)).
```

```
% Predicato per recuperare la transizione in analisi
get_current_transition(Transition) :-
    current_transition(Transition).
```

5.0.3 Contatore

Durante l'analisi del file **aruba.wf** abbiamo trovato nelle transition/2 a differenza del file **monday.wf** che erano transition/3. Per questo motivo abbiamo aggiunto un nuovo predicato che converte i fatti transition/2 in transition/3 aggiungendo arbitrariamente il valore unico per le ripetizioni a ciascuna transition. Codice per gestire il contatore(model_explorator.pl, 17):

```
% Inizializzo contatore usato nel salvataggio dei dati nel file pnml
:- dynamic(counter/1).
counter(0).

% Inizializzo contatore per gli id dei place, delle transition e degli arc nel file pnml
increment_counter :-
    retract(counter(Count)),
    CountPlusOne is Count + 1,
    assert(counter(CountPlusOne)).
```

5.0.4 transition_check

Questo predicato serve a convertire transition/2 in transition/3 aggiungendo un numero, che incrementa ad ogni iterazione, come terzo paramentro(model_explorator.pl, 247):

```
% Questo predicato viene usato nel caso in cui troviamo dei fatti
% transition/2, che vengono convertiti a transition/3 aggiungendo un
% valore unico come terzo elemento
transition_check :-
    get_module(Module),
    clause(Module:Module:transition(FirstInput-FirstOutput, FirstList), true),
    counter(Count),
    increment_counter,
    assert(Module:Module:transition(FirstInput-FirstOutput, FirstList,Count)),
    fail.

transition_check.
```

5.0.5 Inizio delle operazioni

La procedura da cui iniziano tutte le operazioni si chiama **process_file(FileName)**(model_explorator.pl, 40)

```
% Legge il file wf e lo elabora
process_file(FileName) :-
    ask_folder_path(Path),
    atom_concat(Path,"/result.txt",ResultPath),
    use_module(FileName),
    ask_module_name(ModuleString),
    string_to_atom(ModuleString, Module),
    set_module(Module),
    transition_check,
    open(ResultPath, write, Stream),
    do_works(Stream, Path).
```

Analizziamo ora le sue operazioni:

1. ask_folder_path(Path): chiede da terminale di inserire il percorso in cui andare a salvare il file di output(procedura praticamente identica a quella di lettura del modulo)
2. atome_concat(Path,"/result.txt",ResultPath): viene usato per aggiungere il nome del file risultato al percorso inserito in input
3. use_module(FileName): carica il modulo del percorso in input
4. ask_module_name(ModuleString): chiede da terminale il nome del modulo da usare
5. string_to_atom(ModuleString, Module): converte la stringa del modulo in Atomo per poter essere usato durante le operazioni
6. set_module(Module): importa la variable dinamica con l'Atomo appena ottenuto
7. transition_check: viene usato per verificare se le transizioni lette dal modulo in input sono corrette, nel caso in cui non lo fossero vengono convertite nel formato accettato
8. open(Path, write, Stream): crea il file nel percorso specificato e crea lo Stream da usare
9. do_works(Stream, Path): inizia le operazioni di ricerca e analisi delle transizioni

5.0.6 do_works

Questa procedura si preoccupa di trovare tutte le transizioni e inserirle in una lista per poterle analizzare in seguito iterando su di essa, sfrutta la procedura **get_module**(model_explorator, 52).

```
% Inizia esecuzione principale
do_works(Stream, Path) :-
    get_module(Module),
    findall(Module:transition(Input-Output, Cases, Repetition),
            Module:transition(Input-Output, Cases, Repetition),
            ListTransitions), % Gather all transitions into a list
    do_work(Module, Stream, ListTransitions),
    generate_dot_files(Path).
```

Richiama la procedura **do_work(Module, Stream, ListTransitions)** che inizierá il ciclo per verificare tutte le transizioni. In seguito genera i file dot per i grafi.

5.0.7 do_work

Il motivo per cui diamo in input Module é perché ci serve per specificare il modulo nella transizione in input. Questa procedura si occupa di scrivere l'introduzione alla transizione in analisi e poi richiama le procedure per la ricerca delle transizioni da visualizzare. Inoltre itera su se stessa per analizzare tutte le transition in input(model_explorator.pl, 61).

```
% Esegue le operazioni di scrittura su file e recupero delle informazioni delle transition
do_work(Module, Stream, [Module:transition(Input-Output, Cases, Repetition) | Rest]) :-
    set_current_transition(Module:transition(Input-Output, Cases, Repetition)),
    write(Stream, '-----------------------------------------------'),nl(Stream),
    write(Stream, 'Starting transition:'), nl(Stream),
    write(Stream, Input-Output),
```

```
    write(Stream, ','),
    write(Stream, Cases),
    write(Stream, ','),
    write(Stream, Repetition),nl(Stream),
    writeln(Stream, 'You can continue to the following ones:'),
    get_matching_transitions_with_case(Stream, Output-_, Cases, Transitions),
    do_work(Module, Stream, Rest).
```

5.0.8 get_matching_transitions_with_case

In questa procedura avviene l'operazione di selezione delle transizioni che seguono a quella in analisi. Viene sfruttato **findall** per poter trovare le transizioni che:

1. Hanno l'input cercato
2. All'interno della lista dei casi si trova uno dei casi della transizione in analisi

Il primo punto viene ottenuto utilizzando il predicato **contains_element** che ritorna **true** se esiste un elemento in comune tra le due liste in input(model_explorator.pl, 181).

```
% Predicato che ci permette di verificare se
% ci sono elementi in comune tra due liste
contains_element([], _) :-
    false.
contains_element([H|B], List) :-
    member(H, List),
    !.
contains_element([H|B], List) :-
    \+ member(H, List),
    contains_element(B, List).
```

Il secondo punto viene realizzato sfruttando **member**. Tutte le transizioni trovate vengono salvate all'interno della lista **NewTransitions**.

Ora peró non abbiamo ancora finito perché abbiamo trovato tutte le transizioni che combaciano con un solo caso della lista(il primo), dobbiamo analizzare gli altri casi. Per fare questo andiamo a reiterare sulla stessa procedura. Per evitare di ottenere duplicati nella lista(transizioni che condividono piú di un caso) usiamo la procedura **add_to_list_no_duplicates** che leggerá in input la lista delle nuove transizioni,le comparerá con quelle giá trovate(nella prima iterazione non ce ne saranno), e tutte quelle nuove vengono aggiunte alla lista, creando quella nuova che verrá mandata alla prossima iterazione. Tutte le transizioni giá trovate verranno scarate(model_explorator.pl, 67).

```
% Recupera tutte le transizioni che hanno nella lista di input almeno una azione completata
% dalla lista di output (usando il predicato contains_element), identificandole
% come possibile transizioni successive
% Alla fine tutte le transizioni che rispettano i parametri di ricerca sono salvati nella
% lista NewTransitions
% Tutte le transizioni che non sono gia state trovate (e quindi che non sono nella lista
% Transitions), vengono
% aggiunte alla lista finale che si chiama LastTransitions che alla fine verra stampata
get_matching_transitions_with_case(Stream, Input-Output, [Case|Cases], Transitions) :-
    get_module(Module),
    findall(Module:transition(ListInput-Output, TransitionsList, Repetition),
```

```
            (Module:transition(ListInput-Output, TransitionsList, Repetition),
             contains_element(Input,ListInput),
             member(Case, TransitionsList)),
            NewTransitions),
    add_to_list_no_duplicates(NewTransitions, Transitions, LastTransitions),
    get_matching_transitions_with_case(Stream, Input-Output, Cases, LastTransitions).


% Caso base che termina identificando tutte le transizioni che sono
% teoricamente possibili ma
% che non sono mai state osservate dal sistema e stampa su file tutte le
% transizioni trovate
get_matching_transitions_with_case(Stream, Input-Output, [], Transitions) :-
    get_matching_transitions(Input-Output, ExtraTransitions),
    print_safe_list(Stream, Transitions),
    add_to_list_deleting_duplicates(ExtraTransitions, Transitions, NewExtraTransitions),
    writeln(Stream,'Possible transitions that could happen even if the cases do not match:'),
    print_possible_list(Stream, NewExtraTransitions).
```

Il secondo caso rappresenta la situzione in cui abbiamo finito di analizzare tutte le transizioni riguardante quella corrente, quindi stampiamo il risultato. In questa procedura andiamo anche a richiamare **get_matching_transitions** che ci permette di ottenere tutte le transizioni che condividono lo stesso input indipendentemente dai casi, sfruttiamo questa procedura per proporre anche transizioni che potrebbero accadere ma che non sono state coperte dai casi del sistema. Per evitare duplicati, dopo aver trovato tutte le transizioni che condividono l'input con quella in analisi andiamo a togliere quelle giá trovate con la procedura **add_to_list_deleting_duplicates** che date due liste in input restituisce una lista contenente gli elementi della prima senza gli elementi della seconda(eliminando quindi le transizioni valide trovate in precedenza). Infine le stampa.

5.0.9 add_to_list_no_duplicates

Questa procedura riceve in input due lista e ritorna una terza lista che contiene gli elementi della prima lista e della seconda lista senza duplicati(model_explorator.pl, 114).

```
% Questo predicato serve per copiare nella seconda lista tutti
% gli elementi della prima lista che non sono presenti al suo
% interno (Usato per trovare i casi con la corrispondenza senza
% tenere duplicati)
add_to_list_no_duplicates([], List, List).

add_to_list_no_duplicates([H|B], SecondList, Output) :-
    member(H, SecondList),
    !,
    add_to_list_no_duplicates(B, SecondList, Output).

add_to_list_no_duplicates([H|B], SecondList, Output) :-
    add_to_list_no_duplicates(B, [H|SecondList], Output).
```

Questa procedura ha 3 casi:

1. il primo é il caso base in cui la lista di output viene unificata alla seconda
2. il secondo caso é dove verifichiamo se l'emento in testa alla prima lista esiste nella seconda, se esiste semplicemente lo saltiamo e reiteriamo

3. il terzo caso é il caso in cui l'elemento non apparteneva alla lista, ha fatto fallire il secondo caso e quindi in questo lo andiamo ad aggiungere alla testa della seconda lista e poi reiteriamo.

5.0.10 get_matching_transitions

Questa procedura ci permette di trovare le transizioni che convidono lo stesso input,viene usata per trovare le transizioni che potrebbero accadere ma che non sono state analizzate dal sistema. Usa lo stesso meccanismo della procedura **get_matching_transitions_with_case**, ma senza il controllo sulla lista dei casi(model_explorator.pl, 104).

```
% Cerco le transition solo in base alla azione di input
% Basta che almeno una sia presente per essere individuata come possibile condizione
% successiva
get_matching_transitions(Input-Output, Transitions) :-
    get_module(Module),
    findall(Module:transition(ListInput-Output, TransitionsList, Repetition),
            (Module:transition(ListInput-Output, TransitionsList, Repetition),
            contains_element(Input,ListInput)),
            Transitions).
```

5.0.11 add_to_list_deleting_duplicates

Questa procedura é una versione modificata di **add_to_list_no_duplicates**, perché al posto di andare ad aggiungere ad una lista esistente nuovi elementi, va ad aggiungere ad una lista nuova solo gli elementi che non esistono nella seconda lista(nel nostro caso le transizioni che non convidono casi con quella analizzata, permettendoci di individuare quelle possibili ma non analizzate dal sistema)(model_explorator.pl, 117).

```
% Questo predicato serve per creare una nuova lista che
% contiene gli elementi della prima lista meno gli elementi
% della seconda (Usato per vedere i possibili casi extra senza
% corrispondenza)
add_to_list_deleting_duplicates(List, SecondList, Output) :-
    add_to_list_deleting_duplicates(List, SecondList, [], Output).

add_to_list_deleting_duplicates([], _, Temp, Temp).


add_to_list_deleting_duplicates([H|B], SecondList, Temp, Output) :-
    member(H, SecondList),
    !,
    add_to_list_deleting_duplicates(B, SecondList, Temp, Output).

add_to_list_deleting_duplicates([H|B], SecondList, Temp, Output) :-
    add_to_list_deleting_duplicates(B, SecondList, [H|Temp], Output).
```

Possiamo trovare 4 casi:

1. Il primo caso é quello che riceve la prima lista e la seconda lista in input e richiama la procedura con un nuovo parametro utile a unificare alla fine la nuova lista

2. Il secondo caso é il caso base, ovvero quando abbiamo finito gli elementi da confrontare e quindi la nuova lista viene unificata a quella di output

3. Il terzo caso contiene il caso con il controllo che verifica se esiste o no l'elemento nella seconda lista, se esiste va avanti e itera di nuovo, altrimenti passa al quarto caso

4. L'ultimo caso aggiunge l'elemento trovato alla nuova lista e reitera

5.0.12 print_safe_list

Con questo predicato stampiamo una lista in input nello Stream ricevuto(model_explorator.pl, 147).

```
% Predicato che stampa la lista in input e divide con dei trattini
% la formattazione, non indica nessuna proprieta particolare sugli archi
print_safe_list(Stream, []) :-
    write(Stream, '----------'),nl(Stream),
    flush_output(Stream).
print_safe_list(Stream, [H|B]) :-
    get_current_transition(Transition),
    assert(edge(Transition, H,"")),
    writeln(Stream, H),
    print_safe_list(Stream, B).
```

Possiamo trovare 2 casi:

1. Il primo caso é quello che considera la lista vuota e quindi la fine della stampa che viene indicata con dei trattini orizzontali, in seguito viene svuotato anche il buffer altrimenti si rischia di riempirlo e non riuscire piú a stampare su file

2. Il secondo caso é il caso con la lista ancora con elementi da stampare, viene stampata la testa della lista e poi viene reiterato il ciclo.

Oltre a stampare sul file di testo andiamo a creare gli edge che sono utili alla creazione del file dot in seguito. Il predicato include **safe** nel nome perché viene usata per scrivere solamente le transizioni considerate dai casi del sistema.

5.0.13 print_possible_list

Questo predicato é molto simile al precedente, viene usato per scrivere le transizioni possibili ma che non sono state osservate dai casi di sistema, per questo nelle proprietá degli edge aggiunge "style="dashed"," che ci permetterá in seguito di rendere questi archi tratteggiati per poterli distinguere con facilitá (model_explorator.pl, 159).

```
% Questo predicato stampa la lista in input e divide con dei trattini
% la formattazione.
% In particolare questo predicato stampa anche la proprieta che indica lo stile degli archi(t
print_possible_list(Stream, []) :-
    write(Stream, '----------'),nl(Stream),
    flush_output(Stream).
print_possible_list(Stream, [H|B]) :-
    get_current_transition(Transition),
    assert(edge(Transition, H,'style="dashed",')),
    writeln(Stream, H),
    print_possible_list(Stream, B).
```

5.0.14 generate_dot_files

Questo predicato ci permette di generare i file dot che contengono le informazioni necessarie per creare un grafo. Vengono generati due file dal percorso ricevuto in input, il primo avrá come proprietá **ordering=out**, questa proprietá renderizza i nodi e gli archi in base all'ordine di lettura. Il secondo avrá come proprietá **rankdir=LR**, questa proprietá cerca di generare il grafo concentrando i nodi da sinistra verso destra. Usiamo **atom_concat** per poter aggiungere al percorso ricevuto in input i nomi dei file di output.

```
% Questo predicato genera un file dot che contiene tutta la descrizione del grafo.
% Verra scritto con la proprieta ordering=out per la visualizzazione del grafo
generate_dot_files(Path) :-
    atom_concat(Path, "/orderingout.dot", OrderingPath),
    open(OrderingPath, write, OrderingStream),
    write(OrderingStream, "digraph G {\n"),
    write(OrderingStream, "ordering=out;\n"),
    write_edge(OrderingStream),
    write(OrderingStream, "}\n"),
    close(OrderingStream),
    atom_concat(Path, "/rankdirlr.dot", RankdirlrPath),
    open(RankdirlrPath, write, RankdirlrStream),
    write(RankdirlrStream, "digraph G {\n"),
    write(RankdirlrStream, "rankdir=LR;\n"),
    write_edge(RankdirlrStream),
    write(RankdirlrStream, "}\n"),
    close(RankdirlrStream).
```

5.0.15 write_edge

Questo predicato ci permette di scrivere gli edge nei file dot. I file dot possono essere in seguito processati per ottenere una rappresentazione grafica del grafo. Analizziamo ogni edge fino a che la procedura non esamina tutti gli edge presenti in memoria e prosegue nella seconda procedura terminando con successo. Viene usato il predicato **get_transition** che unifica la transizione in input a quella di output permettendoci di ottenere le informazioni al suo interno. Viene usato il predicato **number_of_items_in_common** per verificare quanti casi in comune hanno la transizione di partenza e quella di arrivo, questo valore indicherá il peso dell'arco. Infine scriviamo tutte le informazioni rilevanti delle transizioni nel file(model_explorator.pl, 212).

```
% Questo predicato serve a stampare nel file dot tutti gli edge trovati
write_edge(Stream) :-
    edge(From, To, Properties),
    get_transition(From, Module:transition(FirstInput-FirstOutput,
                                           FirstList, FirstRepetition)),
    get_transition(To, Module:transition(SecondInput-SecondOutput,
                                         SecondList, SecondRepetition)),
    length(FirstList, FirstWeight),
    length(SecondList, SecondWeight),
    number_of_items_in_common(FirstList, SecondList, Count),
    format(Stream, '"~w Weight:~w"->"~w Weight:~w"[~wlabel="~w"];\n',
           [Module:transition(FirstInput-FirstOutput, FirstList, FirstRepetition),
           FirstWeight,
           Module:transition(SecondInput-SecondOutput, SecondList, SecondRepetition),
           SecondWeight,
           Properties,
           Count]),
    fail.
write_edge(_).

% Questo predicato serve a unificare la transizione in input a quella di output
get_transition(Transition, Transition).


% Questo predicato serve a trovare il numero di elementi in comune tra due liste
number_of_items_in_common([], [], 0).

number_of_items_in_common([], _, 0).

number_of_items_in_common(_, [], 0).

number_of_items_in_common([H|B], List, Count) :-
    member(H, List),
    number_of_items_in_common(B, List, PrevCount),
    Count is PrevCount + 1.

number_of_items_in_common([H|B], List, Count) :-
    \+ member(H, List),
    number_of_items_in_common(B, List, Count).
```

Il peso dei nodi invece viene indicato dalla lunghezza della lista dei casi, ovvero per quanti casi la transizione é stata ripetuta nel sistema.

Con questa procedura concludiamo l'analisi del codice di questa sezione.

5.0.16 Descrizione del caso di test con monday.wf

Analizziamo il caso con il file **monday.wf** Questo programma viene caricato in SWI-Prolog usando il comando:

consult('Percorso_fino_al_file/model_explorator.pl').

In seguito viene richiamata la procedura:

process_file('Percorso_fino_al_file/monday.wf').

Alla richiesta del percorso dove salvare il file inserire:

Percorso_fino_a_dove_salvare_i_file_di_output

Alla richiesta del nome del modulo inserire il nome del modulo caricato:
monday

Il programma restituirá true e salverá nel percorso indicato i file. Nel nostro caso nella repository github il file é:
visualizzazione_modello/output_monday

La stessa procedura si ripete con il file aruba, cambiando il nome del modulo in input da **monday** a **aruba** e cambiando il percorso dei file di output per non cancellare quelli generati in precedenza. Nel nostro caso nella repository github il file é:
visualizzazione_modello/output_aruba

I file dot creati possono essere convertiti in una rappresentazione grafica grazie a Graphviz:

1. Assicurarsi di aver installato le librerie di Graphviz dal sito: `https://graphviz.org/download/`
2. Aprire il terminale nel percorso del file da convertire
3. Utilizzare il seguente comando per generare il grafo in formato pdf: dot -Tpdf -o nome_file_output.pdf nome_file_input.dot

Nel nostro caso il comando per generare un file grafo partendo dal file dot ´e stato:

```
dot -Tpdf -o graph_orderingout.pdf orderingout.dot
```

Tutti i grafi pdf basati sui file dot in input sono stati generati e sono presenti nelle rispettive cartelle di output. In base alla complessitá del grado il tempo necessario per il suo render potrebbe variare da alcuni secondi a diversi minuti.
Nel caso di **monday** i grafi sono stati renderizzati in circa 1-2 minuti;
Nel caso di **aruba** entrambi i render hanno richiesto circa 10 minuti, utilizzando un computer con le seguenti specifiche:

1. Processore: i7-8700k 3.70 GHz
2. Ram: 32 GB, 3000 MHz

Occupando circa il 16% della capacitá computazionale del processore durante l'esecuzione. Questo é dato anche dal fatto che il grafo deve essere renderizzato con il peso sugli archi e quindi diventa graficamente piú complesso.

6 Conclusione

Questo progetto ci ha permesso di approfondire il linguaggio Prolog e quindi prendere confidenza con un metodo di programmazione e ragionamento nuovo che ci ha permesso di trovare le soluzioni cercate per risolvere i diversi problemi trovati ragionando da punti di vista differenti.