Relazione del progetto di Ragionamento Automatico
Algoritmo di Chiusura di Congruenza

Jenny Bonato
Matricola: VR431394
Data: 09/01/2019

Introduzione al problema e utilizzo del programma

L'obbiettivo di questo progetto è realizzare un implementazione dell'Algoritmo di Chiusura di Congruenza per la logica del I Ordine libera dai quantificatori. Tale algoritmo utilizza un grafo diretto aciclico opportunamente costruito in cui ogni nodo è un simbolo di costante oppure un simbolo di funzione e ogni arco rappresenta la relazione di essere argomento del nodo padre. Ogni nodo è unico, pertanto è possibile che ci siano diversi padri per uno stesso nodo.

Per utilizzare il seguente programma è sufficiente editare un file di testo e posizionarlo nella cartella `data`. Tale file può avere diverse forme: per ogni riga una uguaglianza o disuguaglianza di simboli di funzione o di costante, per ogni riga un simbolo di funzione o di costante per ogni riga senza predicato d'uguaglianza oppure è anche possibile creare un file di input che sia una combinazione delle due possibilità. È possibile inserire predicati liberi ma, dal momento che non vengono interpretati, vengono considerati esattamente come funzioni. Per eseguire il programma è necessario eseguire il seguente comando:

```
python3 cc.py <file_input>
```

Struttura del Progetto

Il progetto è stato implementato in `python3` ed è strutturato nelle seguenti cartelle: la cartella `doc` in cui è presente tutta la documentazione riguardo il progetto, la cartella `data` in cui sono presenti i file di input del programma, la cartella `test` in cui sono presenti i file di test e la cartella principale in cui si trovano tutti i file che compongono il programma.

Lo sviluppo del programma ha prodotto i seguenti file:

- `cc.py` : in questo file troviamo il `main` del programma. Il programma elabora il file di input e lo passa al `Parser` che costruisce il grafo. Successivamente si occupa di stampare a video il grafo di partenza, fare il `merge` dei nodi che sono in relazione di uguaglianza e ritornarne l'output ovvero se l'insieme è soddisfacibile allora stampa l'albero risultante altrimenti dichiara l'insoddisfacibilità.

- `dag.py` : in questo file possiamo trovare la definizione della classe `DAG`. La classe ha come attributo un dizionario in cui vengono memorizzati i nodi, le chiavi del dizionario sono rappresentate dall'hash del nodo che rappresentano. Per esempio, se nel dag è presente il nodo `f(a)` la chiave che lo rappresenta nel dizionario è esattamente `hash("f(a)")`, questa scelta implementativa verrà meglio spiegata nella prossima sezione. I metodi di tale classe sono quelli dell'algoritmo cc visti a lezione, sono `merge`, `union`, `congruent`, `find`. Tutti questi metodi svolgono operazioni sui nodi del dizionario.

- `node.py` : in questo file possiamo trovare la definizione della classe `Node`. Ogni nodo ha come attributi: `id` rappresentato dall'hash del termine che identificano, `fn` il simbolo di funzione o di costante che implementano, `find` l'id del rappresentante della classe di equivalenza cui appartiene, `ccpar` la lista degli id dei suoi genitori, `args` la lista degli id dei suoi argomenti, `enemies` ovvero la lista degli id dei nodi con cui non può finire nella stessa classe di equivalenza, `friends` questo attributo viene mantenuto nel nodo rappresentante della classe di equivalenza e rappresenta tutti i nodi nella stessa classe.

- `parser.py` : in questo file possiamo trovare la definizione della classe `Parser`. Tale classe ha come attributi il dizionario `nodes`, le cui chiavi sono gli hash dei nodi che ha già costruito e i cui valori sono le stringhe di cui la chiave è hash. Inoltre sono attributi della classe due liste `eq`, `diseq` in cui vengono memorizzate le coppie di id che sono in relazione di uguaglianza o disuguaglianza. Il Costruttore della classe istanzia gli attributi e chiama la funzione `parse_data` e gli passa la lista in input, tale funzione si occupa in primo luogo di chiamare la funzione `division_eq` che divide la lista in input nelle liste di equazioni e disequazioni, nel fare quest'operazione popola il dizionario con tutti i termini coinvolti. In secondo luogo si occupa di chiamare la funzione `build_node` su ogni termine del dizionario `nodes`, tale funzione si occupa di creare il nodo del termine e ricorsivamente anche i nodi dei possibili argomenti del termine. L'identificazione degli argomenti è permessa grazie ad una funzione ausiliaria `find_sons` che elabora il termine alla ricerca dei suoi argomenti e ritorna la lista di tali argomenti. In fine la funzione `parse_data` si occupa di aggiungere i nemici dei nodi popolando le rispettive liste `enemies` di ogni nodo utilizzando le coppie nella lista `diseq`.

Scelte implementative e Euristiche significative

Una delle scelte implementative più sfruttate è quella di utilizzare come id di ogni nodo l'hash del termine che rappresentano. Per esempio il nodo che rappresenta il termine `f(a)` avrà l'id `hash("f(a)")`. L'utilità di questa scelta è dovuta al fatto che la struttura del nodo non tiene traccia del termine che identifica ma mantiene le liste di argomenti e l'identificatore del simbolo di funzione o di costante. Utilizzando gli hash è possibile, durante l'esecuzione, calcolare direttamente l'id del nodo che vogliamo raggiungere. Questa scelta si combina con quella di utilizzare un dizionario per memorizzare i nodi: calcolando l'hash direttamente è possibile accedere al nodo semplicemente con `nodes[id]`, questo permette di lasciare la ricerca e l'estrazione del nodo all'implementazione di python dei dizionari.

Complessità

Test effettuati

Possibili Estensioni

Riferimenti bibliografici

[1] Aaron R. Bradley, Zohar Manna. *The Calculus of Computation. Decision Procedures with Applications to Verification.*.

[2] Daniel Kroening, Ofer Strichman. *Decision Procedures. An Algorithmic Point of View.*