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`. Il file deve avere per ogni riga un'uguaglianza o disuguaglianza di simboli di funzioni oppure di costanti, è possibile introdurre predicati diversi dall'uguaglianza, in tal caso vengono considerati 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.

Un ulteriore scelta implementativa è stata quella di utilizzare le liste `enemies`, `friends`. Entrambe le liste vengono utilizzate per decidere se fare il merge di due nodi, in particolare la lista `enemies` mantiene per ogni classe di equivalenza l'elenco dei nodi con cui non può essere unita mentre la lista `friends` dualmente memorizza per ogni classe di equivalenza la lista di nodi che ne fanno parte. Questa scelta ci permette di ridurre la complessità dell'algoritmo perché posso dichiarare l'insoddisfacibilità dell'insieme appena è necessario fare il merge di due nodi in relazione di disuguaglianza ovvero nella lista `enemies`. Inoltre è molto importante notare che per permettere il merge di due nodi `n1`, `n2` è necessario che ogni nodo nella lista `enemies` di `n1` non sia nella lista `friends` di `n2`, se non avessimo la lista `friends` avremmo bisogno di visitare tutto il grafo alla ricerca dei nodi nella stessa classe di equivalenza.

Abbiamo visto l'utilità di queste liste ma è importante notare che sono anche semplici da mantenere, a occuparsene è la funzione `union`. In questa funzione sono presenti altre scelte implementative che utilizzano proprietà di queste liste. In primo luogo la `union` sceglie quale dei due nodi diventerà il rappresentante della nuova classe di equivalenza in base alla grandezza della lista `friends` più grande. L'idea è che se scelgo il nodo che ha la classe di equivalenza più grande arriverò prima ad un ipotetico fallimento. In seguito all'effettiva unione dei nodi la funzione si occupa di rendere consistente le liste mettendo tutti gli elementi nel rappresentante della classe e togliendole dal nodo che è stato aggiunto. Un'ulteriore semplificazione sulla funzione `union` è quella relativa all'aggiornamento del campo `find`: l'identificazione del nodo identificatore viene svolta da una catena di chiamate della funzione `find`, per ridurre questa catena la funzione `union` si occupa

Input	Tempo di Esecuzione(s)
input1.txt	0.001880
input2.txt	0.001500
input3.txt	0.001568
input4.txt	0.001877
input5.txt	0.001470
input6.txt	0.002199
input7.txt	0.001830
input8.txt	0.002017
input9.txt	0.002057
input10.txt	0.001931
input11.txt	0.002159
input12.txt	0.002045
input13.txt	0.001882
input14.txt	0.001752

Tabella 1: Tabella dei tempi di esecuzione

di cambiare il campo `find` non soltanto del nodo che diventa parte della classe di equivalenza ma anche di tutti quelli che facevano parte della sua lista `friends` ovvero tutti quelli che erano nella sua classe di equivalenza.

Complessità

Test effettuati

In un primo momento per testare il programma sono stati scritti dei test utilizzando la classe `unittest` di python.

Successivamente è stato possibile testarne il funzionamento utilizzando la bibliografia di questa relazione. Gli input che sono stati passati al programma per testarli si trovano nella cartella `data`, riportiamo nella tabella 1 i tempi di esecuzione con i rispettivi input.

Possibili Estensioni

Riferimenti bibliografici

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

diseq./eq.	Soddisfacibile	Insoddisfacibile
0/2	0.001470	-
0/3	0.001830	-
0/5	0.002045	-
0/6	0.002199, 0.001931	-
0/7	0.002159	-
1/1	-	0.001880
1/2	0.001500, 0.002017	0.002057
1/3	0.001882	0.001752
2/3	-	0.001568
3/4	-	0.001877

Tabella 2: Tabella dei tempi di esecuzione

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