I can't get no

deez nuts

99:99:99

Contents

1 Satisfaction

valida vera in tutti i mondi

soddisfacibile vera in alcuni mondi

insoddisfacibile falsa in tutti i mondi

due formule α e β sono *logicamente equivalenti* se α vera $\iff$ β vera , oppure $models(\alpha) \equiv models(\beta)$

1.1 Risultatini

1.1.1 α valida $\iff$ $\neg\alpha$ insoddisfacibile

(codeste sono meta formule, formule sulle formule)

1.1.2 $(\alpha \models \beta) \iff (\alpha \Rightarrow \beta)$ valida

i mondi in cui β sono un sovrainsieme dei mondi in cui α, quindi ogni volta che α allora β, quindi in $\alpha \Rightarrow \beta$ o sono vero vero, o sono falso falso.

inoltre $\alpha \Rightarrow \beta$ non è valida solo quando ci sono casi in cui $\alpha \wedge \neg\beta$, che sono gli stessi casi in cui non vale l'entainment

il fatto che $\alpha \Rightarrow \beta$ sia vero vuol dire che $\alpha \wedge \neg\beta$ è insoddisfacibile, dimostrare che $\alpha \models \beta$ partendo dal dimostrare che $\alpha \wedge \neg\beta$ è impossibile è lo stesso meccanismo delle dimostrazioni per assurdo

2 Algoritmi per entainment

1. Model Checking (DPLL)
2. Theorem Proving (Reduction)

hanno cose in comune

- entrambi lavorano su $\alpha \models \beta \iff \alpha \wedge \neg\beta$ insoddisfacibile
- entrambe lavorano sulla `CNF` (conjunctive normal form), che è una grossa congiunzione di tante disgiunzioni[1] le disgiunzioni in questione si chiamano clausole, sono disgiunzioni di letterali, un letterale è un simbolo prosizione, oppure un simbolo proposizionale negato

```
// ho scritto questo codice per noia
// non so se compila o meno

struct Literal {
    Symbol symbol;
    bool negated;
};

class Assignment {
private:
    std::map<Symbol, bool> mappings = std::map<Symbol, bool>;
```

[1] vale a dire un grosso `and` di tanti piccoli `or`

```
public:
    Assignment(){}
    Assignment(std::map<Symbol, bool> mappings):mappings(mappings){}
    void add_symbol(Symbol sym, bool b) {
        map[sym] = b;
    }
    void remove_symbol(Symbol sym) {
        // boh, sticazzi
    }
    bool value_of(Symbol sym) {
        return mappings[sym];
    }
}

// assignment rappresentato con una
// std::map<Symbol, bool>;

class DisjunctiveClause {
private:
    std::vector<Literal>& literals;
public:
    DisjunctiveClause(std::vector<Literal>& literals):literals(literals)
    ↪  {}

    // le clausole sono disgiunzioni, quindi ritorna true se anche solo
    ↪  una è vera
    bool is_valid_assignment(std::map<Symbol, bool> ass) {
        for(Literal l: literals) {
            if((!l.negated && ass[l.symbol])) ||
                (l.negated && !ass[l.symbol])
                return true;
        }
        return false;
    }
};

class CNF {
private:
    std::vector<DisjunctiveClause>& clauses;
public:

    CNF(std::vector<DisjunctiveClause>& clauses):clauses(clauses){}

    // è una congiunzione di clausole, quindi devono essere tutte vere
    bool is_valid_assignment(std::map<Symbol, bool> ass) {
        for(DisjunctiveClause clause: clauses) {
            if(!clause.is_valid_assignment(ass))
                return false;
```

```
        }
        return true;
    }
};
```

2.1 DPLL

è tipo backtracking ma con euristiche legate a sat

```
def DPLL(clauses, syms, interpretation):
    """
            nei casi di logica proposizionale
            c'era l'algoritmo bovino per l'entainment,
            quello di enumerare tutti i mondi e controllarli tutti
            si era detto che era più \"smart\" usare un albero per farlo
        visto
            che se si usa un albero possiamo fare pruning,
            vale a dire fanculare un assegmanento parziale prima di
        continuarlo
            che è il motivo per cui si usano le cnf, con le cnf possiamo
        mandare subito a fare
            in culo un assegnemto parziale, visto che tutte le clausole
        devono essere vere
            """

    if all_true([interpretation.check(clause) for clause in clauses]):
        return True
    if any_true([interpretation.check(clause) for clause in clauses]):
        return False

    # mo vamo di eruistiche
    # sono documentate sotto

    # simbolo puro
    p,val = find_pure_symbol(clauses, syms, interpretation)
    if p is not None:
        clauses.remove_clauses_containing(p) # aggiunta mia, sticazzi
        syms.remove(p)
        interpretation.add_binding(p, val)
        return DPLL(clauses, syms, interpretation)

    # unit clause
    p,val = find_unit_clause(sym, clauses, interpretation)
    if p is not None:
        syms.remove(p)
        interpretation.add_binding(p, val)
```

```
        return DPLL(clauses, syms, interpretation)

    # qui abbiamo finito le euristiche, si va brutali di backtracking
    p, rest = syms.split_first()

    return (DPLL(clauses, rest, interpretation.add(p, True)) or
            DPLL(clauses, rest, interpretation.add(p, False)))
```

2.1.1 Simbolo puro

e ora andiamo di euristiche, diciamo che ho

$$\{\{A, \neg B\}, \{\neg B, \neg C\}, \{C, A\}\}$$

qui B compare sempre allo stesso modo, quindi o sempre vero o sempre falso

cosa possiamo dire di questa formula, sapendo che B è sempre allo stesso modo? In teoria niente, ma dal punto di vista "di programma" posso confermare che qui se metto B a false, non perdo nulla, se c'è un'interpretazione del `CNF` che lo soddisfa c'è anche un'interpretazione del `CNF` che rende la formula vera

- se ho un assegnamento con B falso, ok
- se ho un assegnamento con B vero, potrei metterlo a falso e resterebbe vero, tanto le clausole sono or, non è che ne cambio da falso a vero cazzo me la rende falsa?

inoltre, sapendo che ho fissato a vere quelle clausole dove c'era B , non mi servono più a un cazzo, posso toglierle

e poi, avendo tolto quelle clausole potrei pure aver creato nuovi simboli puri, *and so on ad infinitum* (se propaga boyse)

2.1.2 Unit clause

una unit clause è una clausola che consiste di un solo letterale, se ho una cosa del genere

$$\{\{A, \neg B\}, \{\neg B, \neg C\}, \{\neg C\}\{C, A\}\}$$

sticazzi, C va messo a `False`

inoltre, visto che poi ho $\{C, A\}$, e ho già fissato C, ora abbiamo che A è l'unica variabile rimasta nella clausola, rendendola una nuova unit clause, e quindi *PROPAGAMUS*

3 CSP to SAT

prendiamo tot simboli A_{iv}che sono

$$A_{iv} = \begin{cases} True & \text{se } X_i \text{ assegnato a } v \\ False & \text{altrimenti} \end{cases}$$

dobbiamo fare comunque in modo che

- ogni X_i sia assegnata a uno e un solo v

  a uno $\bigwedge_{i=1}^{n}(\bigvee_{i=1}^{n} A_{iv})$
  uno solo $\bigwedge_{v \in Vals}(\bigwedge_{i=1}^{n} \bigwedge_{j=1, j \neq i}^{n}(\overline{A_{iv} \wedge A_{jv}}))$

  poi oltre ai suddetti vincoli si aggiungono anche i vincoli del problema

4 Sat

SAT è np completo ma molte istanze di SAT sono facili da risolvere

il sudoku è facile, ma ci sono certe istanze del sudoku che sono dei puttanai, quindi il caso generico è il puttanaio, perchè ricopre anche quei pochi bastardi[2]

5 Modelli di generazione di cazzi

5.1 Uniform Random $k-$SAT

k sta per il numero di letterali in ciascuna clausola

[2] chiamo da reggio emilia