Reti semantiche

Il sistema SNePS

Panoramica e esempi

Sneps

- Rete semantica e sistema di ragionamento
  - Rete semantica *proposizionale*
  - Diversi tipi di inferenza
- Fonti
  - http://www.cse.buffalo.edu/sneps/Manuals/manual271.pdf
  - http://www.cse.buffalo.edu/sneps/Tutorial/

Grafo relazionale vs. Rete proposizionale

Nella rappresentazione di SNEPS (a destra) la proposizione "Snoopy è un cane" è rappresentata da un nodo (M1!).
Nessun arco denota una proposizione x isa (è un) y, come invece avviene nel grafo relazionale (a sinistra).

Applicazioni e caratteristiche

- Utilizzato nella robotica cognitiva
- Solo i nodi sono espressioni ben formate dotate di una semantica
  - In altri sistemi gli archi che uniscono i nodi denotano proposizioni (che esprimono relazioni tra entità)
- Ogni nodo denota un'entità mentale
  - Non ci sono nodi creati per motivi "tecnici"
  - Anche i nodi che rappresentano variabili hanno una semantica composizionale

An Introduction to SNePS 3. Stuart C. Shapiro. Conceptual Structures: Logical, Linguistic, and Computational Issues. *In Lecture Notes in Computer Science,* Volume 1867, 2000, pp 510-524

Software

- Il sistema SNePS è un software dotato delle seguenti funzionalità:
  - Rappresentare la rete: si utilizza un linguaggio utente fatto di comandi che permettono di fare asserzioni (Socrate è un uomo), esprimere regole (Gli uomini sono mortali) e manipolare la rete (creare proposizioni non asserite)
  - Cercare nella rete nodi con certe caratteristiche (gli individui che sono uomini)
  - Inferire nuove asserzioni a partire da quelle esistenti

Input e Output

1. L'utente usa il linguaggio utente per asserire una proposizione

2. Il sistema crea una rete che rappresenta la proposizione
   - Tipicamente, un nodo "proposizione" da cui emanano degli archi
   - La rappresentazione viene creata internamente al sistema e comunicata all'utente per mezzo di una particolare notazione (linguaggio del sistema)
   - Alcune versioni del sistema erano dotate di un output grafico per rappresentare visivamente la rete

3. Un insieme di proposizioni può essere interrogata oppure può diventare la base per inferenze
   - Sia l'estrazione di proposizioni dalla rete che la deduzione di nuove proposizioni avvengono su richiesta dell'utente, cioè quando l'utente digita il comando corrispondente nel linguaggio utente

Tipi di nodo

- I nodi **molecolari** (M*i*) hanno archi uscenti. Rappresentano proposizioni, incluse le regole di ragionamento, oppure individui.
- I nodi **base** (B*i*) non hanno archi uscenti. Rappresentano individui che hanno certe caratteristiche, ma a cui non vogliamo dare un nome.
- I nodi **variabili** non hanno archi uscenti ma rappresentano individui arbitrari o proposizioni, come le variabili logiche.

- Molecular nodes and pattern nodes have arcs emanating from them. Molecular nodes may represent
- propositions, including rules, or "structured individuals." A molecular node that represents a proposition may
- be asserted or unasserted .

- Pattern nodes represent arbitrary propositions or arbitrary structured individuals,
- and are similar to open sentences in predicate logic. Pattern nodes and unasserted molecular nodes are created
- by the build function. Asserted molecular nodes are created by the assert function.

Relazioni

- Le relazioni tra i nodi sono date dagli archi che li uniscono
- Se non specificato diversamente una relazione tra un nodo A e uno B genera un arco diretto da A verso B con il nome della relazione specificata
- Il comando seguente definisce due relazioni, isa e ako

(define isa ako)

Relazioni predefinite

- forall
- exists
- min
- max
- thresh
- ant
- &ant
- cq
- dcq
- arg
- default
- if
- action
- …

- Esistono relazioni (archi) predefiniti nel sistema
- Nella versione più recente i tipi di relazione sono organizzati in una tassonomia

Asserzione

- Il comando **assert** permette di asserire una proposizione
- Ha come argomento una lista di coppie, di cui:
  - Il primo elemento è una relazione, il secondo un nodo o una lista di nodi

(assert member Clyde class elephant)

- Il sistema costruirà un nodo M1 con un arco member che punta all'identificativo Clyde e un arco class che punta un nodo elefante.
  - M1! Significa che il nodo M1 è asserito

Asserzione: esempio

(assert member Clyde class elephant)

M1!
member
class
Clyde
elephant

Il nodo M1! è asserito (sintatticamente, è rappresentato dal ! dopo il nome del nodo).
È possibile che nel sistema siano presente anche delle proposizioni non asserite, cioè non "credute" dal sistema

Contesti

- Un nodo può essere asserito oppure non asserito
- Allo scopo di rappresentare le credenze di più soggetti diversi, il sistema può contenere più contesti
  - Un contesto è formato da ipotesi
  - Un'ipotesi in un contesto è un nodo asserito oppure derivato via inferenza
- Un contesto si crea con il comando:
  :context nodeset context-name

Struttura di un contesto

- Il contesto di lavoro del sistema ha tre componenti:

1. Un nome: per default, il sistema lavora nel contesto corrente (current context)
2. Un insieme di ipotesi (assunzioni che fanno parte del contesto)
3. Un flag che indica se il contesto è consistente

- Un'ipotesi in un contesto è un nodo asserito oppure derivato via inferenza:
  :context nodeset context-name

Esempio di nodo molecolare

Dato il comando (linguaggio utente)

(assert member Snoopy class dog)

Il sistema crea la rete:

(M1! (CLASS DOG) (MEMBER SNOOPY))

L'espressione di cui sopra è la descrizione non grafica con cui si rappresenta la rete

- è simile al rappresentazione che il sistema mantiene internamente della rete.

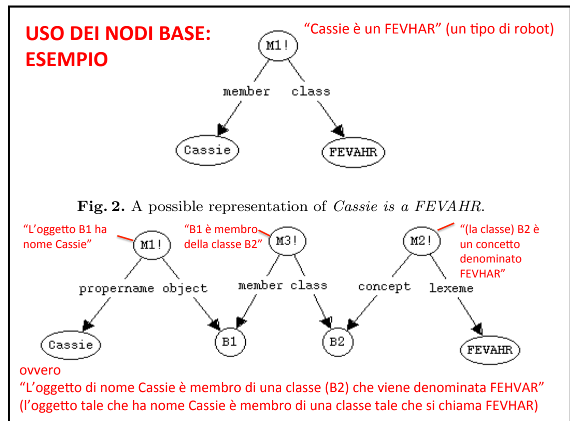
(M1! (CLASS DOG) (MEMBER SNOOPY))

Esempi

- Clyde è un elefante
  (assert member Clyde class elephant)
- Dumbo è un elefante
  (assert member Dumbo class elephant)

Uso dei nodi base

- I nodi base vengono usati per rappresentare individui che soddisfano determinate descrizioni senza dargli un nome
- Non hanno archi uscenti; cioè non hanno informazioni strutturali. Tutto ciò che si sa di loro è ciò che viene asserito.
- Un nodo base potrebbe essere descritto tale l’espressione “un entità tale che …”



Cosa significa?

(assert member Tweety class canary)

(assert object Tweety ability fly)

(assert member Opus class bird)

Significa

(assert member Tweety class canary)
Tweety è un canarino

(assert object Tweety ability fly)
Tweety può volare

(assert member Opus class bird)
Opus è un uccello

Notazione

(assert department CSE
division undergrad
number 116
name "Introduction to Computer Science for Majors II"
credits 4
prerequisites cse115)

(assert member Tweety class canary) (assert object Tweety ability fly)

M2!
M1!
ability
object
member
class
fly
Tweety
canary

M2! = Tweety ha l'abilità di volare
M1! = Tweety è un canarino
"Il canarino Tweety può volare"

Interrogare la rete

- Il comando utente **find** trova uno o più nodi nella rete.
  - In pratica, la rete si comporta come un database di fatti che può essere interrogato descrivendo il tipo di fatti cercato
- Per esempio, il comando seguente:

  (find class elephant)

- Trova tutti i nodi che hanno un arco di tipo classe che esce dal nodo e punta al nodo elefante
  - In pratica trova tutte le proposizioni che esprimo che *qualcosa è un elefante*.

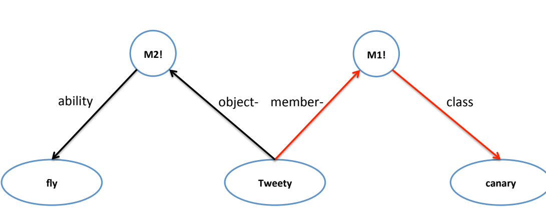
Interrogare la rete con un percorso

- Il comando find permette di specificare uno o più percorsi per trovare i nodi cercati
- (find (member- class) canary (object- ability) fly)
- Trova i canarini (membri della classe canarino) che volano (che hanno l'abilità di volare)

  Nota che: member: arco uscente; member- arco entrante

- Cioè: tutti i nodi che hanno un arco member entrante, collegato (tramite un nodo intermedio) a un arco class, e un arco object entrante, collegato (tramite un nodo intermedio) a un arco ability

(find (member- class) canary (object- ability) fly)



Trova tutti i nodi che hanno un percorso (member- class) verso canary *e* un percorso (object- ability) path to fly.

Esercizio

- Tweety is a canary.
- Tweety can fly.
- Opus is a bird.

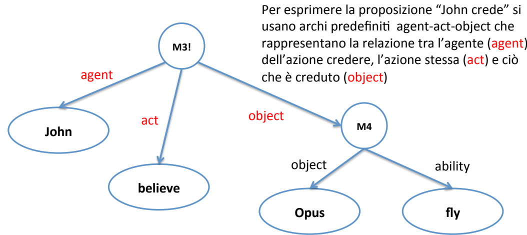
Rappresentazione delle credenze

(assert agent John
act believe
object (build object Opus ability fly))

- Attenzione: il sistema non crede che Opus possa volare ma
- Crede che John creda che Opus possa volare!
- Con un *find* posso scoprire cosa il sistema crede che John creda…

(assert agent John act believe object (build object Opus ability fly))

Per esprimere la proposizione "John crede" si usano archi predefiniti agent-act-object che rappresentano la relazione tra l'agente (agent) dell'azione credere, l'azione stessa (act) e ciò che è creduto (object)



M3 = "John crede M4" (John è *l'agente dell'azione* di credere *l'oggetto* M4)
M4 = "Opus può volare"
"John crede che Opus possa volare"
Importante: M4 non è asserito perché è una credenza di John non del sistema!

Deduzioni

- Il comando deduce permette di cercare proposizioni che non sono direttamente asserite dal sistema ma che il sistema è in grado di inferire
- (deduce member $x class canary)
- Trova tutti gli x che appartengono alla classe dei canarini ($ indica una variabile)

Inferenze in Sneps

- Le inferenze si fanno in tre in modi

1. Riduzione
2. Inferenze basate su percorsi
3. Regole

Riduzione

- La riduzione consiste nel dedurre da un grafo una porzione contenuta in esso:
- (assert agent john act gives object book-1 recipient mary)
  - John da' un libro a Mary
- (deduce agent john act gives object book-1)
  - John da' un libro

Esempio

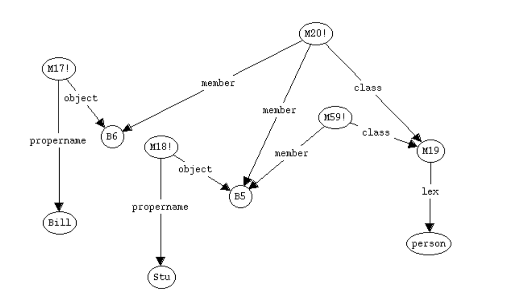


Fig. 7. Node M20!, denoting the proposition that Stu and Bill are people, implies M59! denoting the proposition that Stu is a person, by wire-based inference.

M59! Corrisponde alla sola proposizione "Stu è una persona" (o meglio "l'entità di nome Stu appartiene alla classe chiamata person")

Inferenze: path

- Path-based inference allows an arc between two nodes to be inferred from the presence of a path of arcs
- between them. The various versions of find as well as deduce will use any path-based inference rules that
- have been declared.

Inferenze basate su percorsi

- E' possibile stabilire che un percorso fatto di certe relazioni è uguale a una relazione
- Si usa il comando **define-path**
- Per definire un percorso si usa il comando **compose** che si può pensare come comporre una nuova relazione (arco) basandosi su un percorso di relazioni
  - In pratica, si dice che un certo percorso (o meglio ogni percorso con certe caratteristiche) è uguale a un singolo arco

Esempio: la relazione isa transitiva

- **Kstar** sta per "zero o più occorrenze" di una certa relazione

  (**define-path** isa (**compose** isa
  (**kstar** (**compose** object- isa))))

- Si legge: isa è guale a isa *composto con* un percorso di zero o più percorsi ottenuti da un object- *composto con* un isa
- In base a questa deifinzione

  isa
  isa object- isa
  …
  È sempre isa

esempio

- (define object isa has)
- (describe (assert object elephant isa animal))
- (describe (assert object circus\ elephant isa elephant))
- (describe (assert object Dumbo isa circus\ elephant))
- (describe (assert object Clyde isa elephant))
- (describe (assert object bird isa animal))
- (describe (assert object Tweety isa bird))
- (describe (assert object animal has head))
- (describe (assert object head has mouth))
- (describe (assert object elephant has trunk))
- (describe (assert object trunk isa appendage))

Cosa significa?

```
(define-path
    has2 (compose (kstar (compose isa object- ))
                  has (kstar (compose object- has))
                  (kstar (compose object- ! isa))))
```

Has2 significa che X *has2* Y if: X isa A and A has B and B isa Y

Dumbo *has2* la proboscide se Dumbo è un elefante che ha un certo organo che è una proboscide.

Regole di ragionamento

- Il terzo modo per ragionare è dato dalle regole
- Le regole sono definite dall'utente in modo arbitrario
  - Questo è il loro punto debole!
- Fanno parte della base di conoscenza
- Permettono di aumentare drasticamente la conoscenza contenuta nella rete, perché aggiungono nuove proposizioni non implicite nella rete

Regole per agire

- Then, we will give a rule that says the way to greet a person is to sayHi , then say the person's name (and
- assert that Stu and Bill are people).
- (M1! (FORALL V1) (ANT (P1 (CLASS PERSON) (MEMBER V1)))
- (CQ
- (P5 (ACT (P2 (ACTION GREET) (OBJECT1 V1)))
- (PLAN
- (P4 (ACTION SNSEQUENCE) (OBJECT1 SAYHI)
- (OBJECT2 (P3 (ACTION SAY) (OBJECT1 V1)))))))
- (M1!)

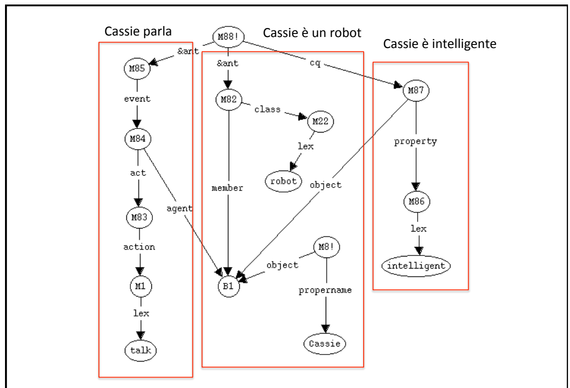


Fig. 5. M88! is a rule node denoting the proposition, *If Cassie is a robot that talks, then Cassie is intelligent.*

Se Cassie è un robot che parla allora Cassie è intelligente

- Antecedente 1 (M85), Cassie parla: in un evento (M84) B1 è l'agente dell'esecuzione (act) dell'azione (M83) denominata parlare (lex talk)
- Antecedente 2 (M82), Cassie è un robot: B1 è membro della classe (M22) denominata (lex) robot e ha come nome proprio (propername) Cassie (quest'ultimo fatto è asserito)
- Conseguente (M87), Cassie è intelligente: B1 (Cassie) ha la proprietà (M86) di essere intelligente (denominate "intelligent".

Esempio

Marcus was a man.
(assert member Marcus class man)

Marcus was a Pompeian.
(assert member Marcus class Pompeian)

Cesare era un tiranno
(assert member Caesar class ruler)

Regola #1

"All Pompeians were Romans"

- Per asserire questa regola è necessario avere i seguenti archi:

- **forall** quantificatore universale
- ant antecedente di una regola if-then
- cq conseguente della regola

```
(assert forall $p
        ant (build member *p class Pompeian)
        cq  (build member *p class Roman))
```

Regola #2

```
(assert forall $m
        ant (build   member *m class man)
        cq  (build   arg1 *m
                     rel loyal\ to
                     arg2
                     (build skolem-function liege\ of
                        arg1 *m))))
```

Per ogni m, se m è un uomo (man), allora m è leale al sovrano (liege) di m (esistenziale skolemizzato)

Esprimere AND, NOT, OR

- In Sneps si utilizza l'espressione
  min 1 max 1 arg

  per indicare che esattamente una in un insieme di proposizioni è vera (corrisponde a XOR)
  - Ci sono due argomenti (arg), almeno uno è vero, al massimo uno è vero
- min 0 max 0

  Per negare una proposizione
  - C'è un solo argomento e nessuno è vero

Regola #3

```
(assert forall $r
       ant (build member *r class Roman)
       cq  (build min 1 max 1
               arg ((build arg1 *r
                        rel loyal\ to
                        arg2 Caesar)
                     (build arg1 *r
                        rel hate
                        arg2 Caesar))))
```

Tutti i romani erano fedeli a Cesare (arg 1) oppure lo odiavano (arg2)

Regola #4

```
(assert forall ($ppl $rlr)
        &ant ((build member *ppl class person)
              (build member *rlr class ruler)
              (build arg1 *ppl
              rel try\ to\ assassinate
              arg2 *rlr))
        cq (build min 0 max 0
            arg (build arg1 *ppl
                rel loyal\ to
                arg2 *rlr)))
```

Le persone assassinano solo i sovrani a cui non sono fedeli
Ovvero
Per tutte le persone *ppl* e per tutti i sovrani *rlr*
Se una *ppl* tenta di assassinare *rlr*, allora *ppl* non è leale al *rlr*

Ultime asserzioni e domanda

- Marcus tried to assassinate Caesar.

```
(assert arg1 Marcus
        rel try\ to\ assassinate
        arg2 Caesar))
```

- Was Marcus loyal to Caesar?

```
(deduce arg1 Marcus rel loyal\ to arg2 Caesar)
```

Esercizio: tutti i cani sono animali domestici ("pet")

```
(assert forall $dog1
ant (build member *dog1 class dog)
cq (build member *dog1 class pet))

(assert forall (*dog1 $cat1)
&ant ((build member *dog1 class dog)
(build member *cat1 class cat))
cq (build agent *dog1 act hates object *cat1))
```

Note:
$ crea una variabile; * serve per riferirsi a una variabile (quindi precede il nome di una variabile già creata)
Build è uguale ad assert tranne che crea un nodo non asserito (senza !)

Esercizio: Creare il grafo per entrambe le regole, per la seconda scrivere anche la regola in linguaggio naturale