FOL
A graph implementation of First Order Logic

M Toussaint

May 19, 2015

1 Graph basics

Our graph syntax is a bit different to standard conventions. Actually, our graph is a hyper graph: nodes can play the role of normal nodes, or hypernodes (=edges or factors/cliques) that connect other nodes. At the same time our graph is a typed dictionary: every node has a set of keys (or tags, to retrieve the node by name) and a typed value (every node can be of a different type).

- A graph is a set of nodes
- Every node has three properties:
  - A tuple of **keys** (=strings)
  - A tuple of **parents** (=references to other nodes)
  - A typed **value** (the type may differ for every node)

The ascii file format of a graph clarifies this:

```
## a trivial graph
x                 # key=x, value=true, parents=none
y             # key=y, value=true, parents=none
(x y)             # key=none, value=true, parents=x y
(-1 -2)             # key=none, value=true, parents=the previous and the y-node

## a bit more verbose graph
node A{ color=blue }                # keys=node A, value=<Graph>, parents=none
node B{ color=red, value=5 }        # keys=node B, value=<Graph>, parents=none
edge C(A,B){ width=2 }                   # keys=edge C, value=<Graph>, parents=A B
hyperedge(B C) = 5                   # keys=hyperedge, value=5, parents=B C

## standard value types
a=string        # MT::String (except for keywords 'true' and 'false' and 'Mod' and 'Include')
b="STRING"        # MT::String (does not require a '=')
c='file.txt'        # MT::FileToken (does not require a '=')
d=-0.1234        # double
e=[1 2 3 0.5]        # MT::arr (does not require a '=')
f=(c d e)        # MT::Array<*Node> (list of other nodes in the Graph)
g                      # bool (default: true)
h=true                 # bool
i=false                 # bool
j={ a=0 }        # sub-Graph (special: does not require a '=')

## parsing: = {..} (..) , and \n are separators for parsing key-value-pairs
b0=false b1 b2, b3() b4   # 4 booleans with keys 'b0', 'b1 b2', 'b3', 'b4'
k={ a, b=0.2 x="hallo"          # sub-Graph with 6 nodes
  y
  z()=filename.org x }

## special Node Keys

## merging: after reading all nodes, the Graph takes all Merge nodes, deletes the Merge tag, and calls a merge()
## this example will modify/append the respective attributes of k
Merge k { y=false, z=otherString, b=7, c=newAttribute }

## including
```

```
Include = 'example_include.kvg'   # first creates a normal FileToken node then opens and includes the file directly

## referring to nodes (constants/macros)
macro = 5
val=(macro) # *G["val"]->getValue<double>() will return 5

## any types
trans=<T t(10 0 0)>  # 'T' is the tag for an arbitrary type (here an ors::Transformation) which was registered somewhere in the code using the registry()
                     # (does not require a '=')

## strange notations
a()           # key=a, value=true, parents=none
()           # key=none, value=true, parents=none
[1 2 3 4]  # key=none, value=MT::arr, parents=none
[2 3 4]
[4 6]
```

A special case is when a node has a Graph-typed value. This is considered a **subgraph**. Subgraphs are sometimes handled special: their nodes can have parents from the containing graph, or from other subgraphs of the containing graph. (When deep copying graphs they requre special care.)

2 Representing KB as a Graph

We represent everything, a full knowledge base (KB), as a graph:

- Symbols (both, constants and predicate symbols) are nil-valued nodes. We assume that they are declared in the root scope of the graph
- A grounded literal is a tuple of symbols, for instance `(on box1 box2)`. Note that we can equally write this as `(box1 on box2)`. There is no need to have the 'predicate' first. In fact, the basic methods do not distinguish between predicate and contant symbols.
- A universal quantification $\forall X$ is represented as a scope (=subgraph) which first declares the logic variables as nil-valued nodes as the subgraph, then the rest. The rest is typically an implication, i.e., a rule. For instance

  $$\forall XY \ p(X,Y) q(Y) \Rightarrow q(X)$$

  is represented as `{X, Y, { (p X Y) (q Y) } { (q X) }` where the precondition and postconditions are subgraphs of the rule-subgraph.

Here is how the standard FOL example from Stuart Russell's lecture is represented:

```
Constant M1
Constant M2
Constant Nono
Constant America
Constant West

American
Weapon
Sells
Hostile
Criminal
Missile
Owns
Enemy

STATE {
(Owns Nono M1),
(Missile M1),
(Missile M2),
(American West),
(Enemy Nono America)
}
```

```
Query { (Criminal West) }

Rule {
x, y, z,
{ (American x) (Weapon y) (Sells x y z) (Hostile z) }
{ (Criminal x) }
}

Rule {
x
{ (Missile x) (Owns Nono x) }
{ (Sells West x Nono) }
}

Rule {
x
{ (Missile x) }
{ (Weapon x) }
}

Rule {
x
{ (Enemy x America) }
{ (Hostile x) }
}
```

2.1 Valued predicates

By default all tuples in the graph are boolean-valued with default value true. In the above example all literals are actually true-valued. A rule `{X, Y, { (p X Y) (q Y) } { (q X)! }` means $\forall XY\ p(X,Y)q(Y) \Rightarrow \neg q(X)$. If in the KB we only store true facts, this would 'delete' the fact `(q X)!` from the KB (for some X).

As nodes of our graph can be of any type, we can represent predicates of any type, for instance `{X, Y, { (p X Y) (q Y)=3 } { (q X)=4 }` would let $p(X)$ be double-typed.

3 Methods

The most important methods are the following:

- Checking whether **two facts are equal**. Facts are grounded literals. Equality is simply checked by checking if all symbols (predicate or constant) in the tuples are equal. Optionally (by default), it is also checked if the fact values are equal.
- Checking whether **a fact equals a literal+substitution**. The literal is a tuple of symbols, some of which may be first order variables. All variables must be of the same scope (=declared in the same subgraph, in the same rule). The substitution is a mapping of these variables to root-level symbols (predicate of constant symbols). The methods loops through the literal's symbols; whenever a symbol is in the substitution scope it replaces it by the substitution; then compares to the fact symbol. Optionally (by default) also the value is checked for equality.
- Check whether **a fact is directly true in a KB (or scope)** (without inference). This uses the graph connectivity to quickly find any KB-fact that shares the same symbols; then checks these for exact equality.
- Check whether **a literal+substitution is directly true in a KB** (without inference).
- Given a single literal with only ONE logic variable, and a KB of facts, **compute the domain** (=possible values of the variable) for the literal to be true. If the literal is negated the $D \leftarrow D \setminus d$, otherwise $D \leftarrow D \cup d$ if the d is the domain for true facts in the KB. [TODO: do this also for multi-variable literals]

- **Compute the set of possible substitutions for a conjunction of literals** (typically precondition of a rule) to be true in a KB.
- **Apply a set of 'effect literals'** (RHS of a rule): generate facts that are substituted literals

Given these methods, forward chaining, or forward simulation (for MCTS) is straightforward.