Modeling Machine Learning and Data Mining Problems with $FO(.)^{IDP}$

Student ID: 111493647
Aishwarya Danoji

Final Project report: Computing with Logic
CSE 505

Instructor: Dr.Paul Fodor

13th December, 2017

Stony Brook University, Department of Computer Science

1 Introduction:

The project deals with using FO(.) and IDP framework for modeling and solving machine learning and data mining task. There exists some problems in machine learning and data mining which follow no standard algorithm for solution. These problems can be abstracted as graph problems and are NP complete. The IDP framework combines a declarative specification in $FO(.)^{IDP}$, with imperative manipulation of the specification via Lua scripting language to solve such problems.

Three problems implemented in IDP-Web IDE are as follows:
1. **Stemmatology** - the humanistic discipline that attempts to reconstruct the transmission of a text (especially a text in manuscript form) on the basis of relations between the various surviving manuscripts (sometimes using cladistic analysis). OR A domain of philology concerned with the relationship between surviving variant version of text.
2. **Minimum Common Sub-graph of partially labeled trees** - The task is about a problem within biology where phylogenetic trees are used to represent evolution if species.
3. **Learning deterministic finite state automata.**

2 Goals:

- Implement Discrete Finite Automata in IDP - Web IDE.
- Extended Goals: Identify and implement other machine Learning and data analysis problems that can be solved using FO(.)IDP.
- Implement Stemmatology in IDP -IDE using Lua Programming language to load the input stemma into structure.

- Implement Finding Minimum Common Supergraph in IDP - Web IDE .

3 Implementation:

3.1 Learning deterministic finite state automata

Definition 1 : Given a pair of finite sets of positive example strings S+ and negative example strings S-, (the input sample), the goal of DFA identification (or learning) is to find a (non-unique) smallest DFA A that is consistent with S = S+, S-, i.e., every string in S+ is accepted, and every string in S- is rejected by A. Typically, the size of a DFA is measured by $\|Q\|$, the number of states it contains.

3.1.1 IDP Solution

- **Version 1**
- The types state, label, the function trans, and the predicates acc and rej describe the given input samples (and hence the APTA).
- The states of the resulting automaton are elements of the type color.
- Its transitions are described by the function colorTrans (partial function). To construct a complete DFA from the result, colorTrans has to be made total by mapping the missing transitions to a hidden sink state.
- The function colorOf maps the states of the APTA on the states (colors) of the final automaton.
- Finally, the predicate accColor describes the accepting states of the resulting automaton.
- The theory expresses two constraints on accColor: accepting states of the APTA must and rejecting states cannot be mapped to an accepting state of the final automaton. The third constraint states that each transition on the APTA induces a transition between colors.
- The term nbColorsUsed counts the number of states (colors) of the resulting automaton and is used for minimization.
  Instead of minimizing the number of states, one could as well minimize other properties such as the number of transitions, depth of the model, the size of loops, etc. They are also easy to formalize in FO()IDP as shown in Figure 1.
- **Version 2**
- To improve the performance ,the redundant constraint can be decompiled as shown in Figure2.This formula can be derived from the formula in previous theory together with the fact that colorOf is a total function.

```
vocabulary dfaVoc {
type state // states used in APTA
type label // symbols triggering transitions
type color // available states for resulting automaton
partial trans(state , label ): state // transitions of APTA
acc(state) // accepting states of APTA
rej ( state ) // rejecting states of APTA
colorOf(state): color // fixed in input for colors in clique
// the resulting automaton :
partial colorTrans(color , label ): color // transitions of DFA
accColor(color) // accepting states
}
theory dfaTheory : dfaVoc {
! x : acc(x) => accColor(colorOf(x)).
! x : rej(x) => ~accColor(colorOf(x)).
    // trans induces colorTrans :
! x l z: trans(x,l)=z => ! i j : colorOf(x)=i & colorOf(z)=j => colorTrans(i , l)=j .
}

structure S:dfaVoc {

    state={a;b;abaa;bb;abb;abaaaaa;aba;bba}
    label={a;b}
    accColor={Blue}
    color={Red;Blue;Green;Yellow}
    acc={a;abaa;bb}
    rej={abb;b}

}

term nbColorsUsed:dfaVoc{
    #{x:(?y :colorOf(y)=x)}

}

procedure main ( ) {
    stdoptions.verbosity.grounding = 1
    stdoptions.verbosity.solving = 2
    print(minimize(dfaTheory,S,nbColorsUsed ) [1] )
}

```

Figure 1: Complete IDP code for Learning DFA

```
vocabulary dfaVoc {
type state // states used in APTA
type label // symbols triggering transitions
type color // available states for resulting automaton
partial trans(state , label ): state // transitions of APTA
acc(state) // accepting states of APTA
rej ( state ) // rejecting states of APTA
colorOf(state): color // fixed in input for colors in clique
// the resulting automaton :
partial colorTrans(color , label ): color // transitions of DFA
accColor(color) // accepting states
}
theory dfaTheory : dfaVoc {
! x : acc(x) => accColor(colorOf(x)).
! x : rej(x) => ~accColor(colorOf(x)).
    // trans induces colorTrans :
 !v l w: trans(w,l)=v =>!j: colorTrans(colorOf(w),l)=j => colorOf(v)=j.
}

structure S:dfaVoc {

    state={a;b;abaa;bb;abb;abaaaaa;aba;bba}
    label={a;b}
    accColor={Blue}
    color={Red;Blue;Green;Yellow}
    acc={a;abaa;bb}
    rej={abb;b}

}

term nbColorsUsed:dfaVoc{
    #{x:(?y :colorOf(y)=x)}

}

procedure main ( ) {
    stdoptions.verbosity.grounding = 1
    stdoptions.verbosity.solving = 2
    print(minimize(dfaTheory,S,nbColorsUsed ) [1] )
}
```

Figure 2: Complete IDP code for Learning DFA :Redundant constrain

3.2 Machine Learning problem

3.2.1 Shortest Path problem

Definition 2 : Given a graph G(E,V), here E =edges and V= vertices, find the shortest path in a graph from one vertex to another. "Shortest" may be least number of edges, least total weight, etc.

IDP Solution

- **Version 1**
- The vocabulary consists of a single type, two constants and three predicates.
- The structure specifies the given graph: the interpretation of the type node and of the predicate edge(node,node), as well as the constants from and to which identify the begin and endpoint of the path searched for.
- The predicate edgeOnPath(node,node) is used to represent the edges that participate in a shortest path. It provides the base case of the transitive relation reaches(node,node) which is defined in the theory component.Here we use the most basic definition for the transitive closure: we join the reaches relation with itself.As shown in Figure 3.
- **Version 2**
- When the performance matters or the instances are that large that the grounding hardly fits in memory, then we modify the theory section for the shortest path problem as shown in Figure4.
- Apart from the risk of entering an infinite loop in version 1, reaches(x,z) can have many more solutions than edgeOnPath(x,z) and hence, the search space for Prolog's proof procedure can be substantially larger.
- Comparing the runtime of both versions reveals that the heuristics of the version 1 do not compensate for the larger search space and that the version 2 is substantially faster.
- Here although the performance has improved quite a bit, we see that it is still rapidly increasing with the size of the graph.
- **Version 3**
- The grounding in version 2 has n^2 atoms reaches(n1,n2), expressing whether n1 and n2 are connected while we are only interested in paths going from from to to.
- Therefore we can better use a unary reachable predicate and define which points are reachable from from.
- The new version of the vocabulary and the theory are shown in Figure 5.

```
vocabulary sp_voc{
    type node
    start:node
    to:node
    edge(node,node)
    edgeOnPath(node,node)
    reaches(node,node)
}

theory sp_theory1:sp_voc {
    reaches(start,to). // (2)
    ! x y : edgeOnPath(x,y) => edge(x,y).
    {
        !x y:reaches(x,y) <- edgeOnPath(x,y) | edgeOnPath(y,x).
        !x y:reaches(x,y) <- ?z: reaches(x,z) & reaches(z,y).

    }
}

structure sp_struct:sp_voc {
    node = {A..D} // shorthand for A,B,C,D
    edge = {A,B; B,C; C,D; A,D} // `; ' separated list of tuples
    start = A
    to = D
}

term lengthOfPath:sp_voc{
    #{x,y:edgeOnPath(x,y)}
}

procedure main() {
stdoptions.verbosity.grounding = 1
stdoptions.verbosity.solving = 2
sol = minimize(sp_theory1,sp_struct,lengthOfPath)[1]
if (sol == nil) then
        print ("No models exist .\n")
else
        print (sol)
end
}
```

Figure 3: Complete IDP code1 for finding Shortest path in un-weighted graph

- **OUTPUT:**
- For a small graph the output for all the 3 versions shown above is the same as shown in Figure 6. When we have a very large graph with many nodes there is a significant difference in the execution time of the 3 versions.

3.3 Stemmatology

Stemmatology is the branch of study concerned with analyzing the relationship of surviving variant versions of a text to each other, especially so as to reconstruct a lost original.
Definition 3 (Color-connected): Two nodes x and y in a colored CRDAG are color-connected if a node z exists (z can be one of x and y) such that there is a directed path from z to x, and one

```
vocabulary sp_voc{
    type node
    start:node
    to:node
    edge(node,node)
    edgeOnPath(node,node)
    reaches(node,node)
}

theory sp_theory1:sp_voc {
    reaches(start,to). // (2)
    ! x y : edgeOnPath(x,y) => edge(x,y). // (1)
    {
        ! x y : reaches(x,y) <- edgeOnPath(x,y) | edgeOnPath(y,x).
        ! x y: reaches(x,y) <- ?z:  edgeOnPath(x,z) & reaches(z,y).
    }

}

structure sp_struct:sp_voc {
    node = {A..D} // shorthand for A,B,C,D
    edge = {A,B; B,C; C,D; A,D} // `; ' separated list of tuples
    start = A
    to = D
}

term lengthOfPath:sp_voc{
    #{x,y:edgeOnPath(x,y)}
}

procedure main() {
stdoptions.verbosity.grounding = 1
stdoptions.verbosity.solving = 2
sol = minimize(sp_theory1,sp_struct,lengthOfPath)[1]
if (sol == nil) then
        print ("No models exist .\n")
else
        print (sol)
end
}
```

Figure 4: Shortest Path IDP code2:Using different definition for reaches

```
vocabulary sp_voc{
    type node
    start:node
    to:node
    edge(node,node)
    edgeOnPath(node,node)
    reacheable(node)
}

theory sp_theory1:sp_voc {

    ! x y : edgeOnPath(x,y) => edge(x,y). // (1)
    {
        reacheable(start). // (2)
        ! x y: reacheable(y) <- edgeOnPath(x,y) & reacheable(x).
    }
    reacheable(to).
}

structure sp_struct:sp_voc {
    node = {A..D} // shorthand for A,B,C,D
    edge = {A,B; B,C; C,D; A,D} // `; ' separated list of tuples
    start = A
    to = D
}

term lengthOfPath:sp_voc{
    #{x,y:edgeOnPath(x,y)}
}

procedure main() {
stdoptions.verbosity.grounding = 1
stdoptions.verbosity.solving = 2
sol = minimize(sp_theory1,sp_struct,lengthOfPath)[1]
if (sol == nil) then
        print ("No models exist .\n")
else
        print (sol)
end
}
```

Figure 5: Shortest Path IDP code3: A unary reachable relation instead of the binary reaches relation.

```
structure  : sp_voc {
  node = { "A"; "B"; "C"; "D" }
  edge = { "A","B"; "A","D"; "B","C"; "C","D" }
  edgeOnPath = { "A","D" }
  reaches = { "A","A"; "A","D"; "D","A"; "D","D" }
  start = "A"
  to = "D"
}
```

Figure 6: Output for Shortest path problem

from z to y, and all nodes on these paths (including z, x, y) have the same color.
Given a partially colored CRDAG, the color-connected problem is to complete the coloring such that every pair of nodes of the same color is color-connected.

3.3.1 IDP Solution

- In Figure 7 the vocabulary part introduces two types (manuscript and color), two functions and one predicate.
- The function colorOf maps a manuscripts to its color and the function sourceOf maps a color to the manuscript that is the source of the feature.
- The predicate copiedBy is used to represent the CRDAG of the stemma in the input structure.
- The theory part compactly represents the color-connectedness property by a single constraint: when the source of the color of a manuscript (x) is not equal to the manuscript itself then there must exist a manuscript (y) with the same color that has been copied by x.
- The Lua code of the procedure process (omitted, 60 lines) processes the stemma data and builds the input structure for copiedBy. It then iterates over the features, partially builds the structure for the function colorOf and calls the procedure check, passing all structures in the variable feature.
- The latter procedure calls the model expansion and returns the result to process which reports them to the user.

3.4 Minimum common supergraphs of partially labelled trees

We are given N number of trees. Our task is to find minimal supergraph that represents each of the individual trees.
Definition 4 (Common supergraph of partially labeled n-graphs). Given is a set S of n names and a set of graphs {G1, G2, . . . , Gt} where each graph Gi = (V, Ei, Li) has n vertices and is partially labeled with an injective function $Li : V -> S$. A graph (S, ES) is a common supergraph of {G1, G2, . . . , Gt} if there exists, for each i, a bijection$L'i : V -> S$ that extends Li and such that, for each edge {v, w} of Ei: {L'i(v), L'i(w)} ∈ ES.
A minimum common supergraph (S, ES) is a common supergraph such that $\|ES'\| \geq \|ES\|$ for all common supergraphs (S, ES).

3.4.1 IDP Solution

- Find a minimum common supergraph (MCS) for every pair of trees.
- Pick the smallest MCS (say G) and remove the two trees that are the input for G.

```
vocabulary V {
  type manuscript
  type color
  copiedBy(manuscript,manuscript)
  colorOf(manuscript): color
  sourceOf(color): manuscript
}
theory T : V {
  ! x : x ~= sourceOf(colorOf(x))
     => ? y : copiedBy(y,x) & colorOf(x) = colorOf(y).
}
procedure check(feature) {
  return sat(T,feature) // checks existence of a model
}
procedure process(stemmafilename,samplefilename) {
read the stemma data and build a structure for copiedBy
for each feature {
   read the given colors and build a partial structure for colorOf
   call check(feature)
   report the results }
}
```

Figure 7: IDP Solution for stemmatology

- Find an MCS between G and every remaining tree.
- Replace G by an MCS with minimum size, remove the tree that is the input for this MCS and go back to step 3 if any tree remains.
  Steps 1 and 3 of this simple procedure are performed by IDP using a model very similar to that of Figure 8.

3.5 Project links :

3.5.1 IDP WEB-IDE links:

- **For DFA version 1:**http://dtai.cs.kuleuven.be/krr/idp-ide/?src=0ae0c81e9040b3082f19b6cab119414a
- **For DFA version 2:**http://dtai.cs.kuleuven.be/krr/idp-ide/?src=18030fa7cc413e5f90a350c67cd18460
- **For Shortest path version 1:**http://dtai.cs.kuleuven.be/krr/idp-ide/?src=a58b56157b343ddb11e3a890f2fb
- **For Shortest path version 2:**http://dtai.cs.kuleuven.be/krr/idp-ide/?src=3cd2b54d0f31bb457656b80095a
- **For Shortest path version 3:**http://dtai.cs.kuleuven.be/krr/idp-ide/?src=aa11788a5c755c90f2da8ee7a259

```
vocabulary CsPltVoc {
  type tree
  type vertex
  type name // Isomorphic to vertex
  edge(tree,node,node)  // trees, given in input structure
  arc(name,name) // The induced network
  label(tree,node): name // the labeling,
                   //partially given in the input structure
}
theory CsPltTheory: CsPltVoc {
  { // induced network
    arc(label(t,x),label(t,y)) <- edge(t,x,y) &
                                  label(t,x) < label(t,y).
    arc(label(t,x),label(t,y)) <- edge(t,y,x) &
                                  label(t,x) < label(t,y).
  }
  ! t c : ?1 n : label(t,n) = c. // label function is bijective
}
term SizeOfSupergraph: CsPltVoc { #{ x y : arc(x,y) } }
procedure main() {
  print(minimize(CsPltTheory,CsPltStructure,SizeOfSupergraph)[1])
}
```

Figure 8: IDP Solution for Finding Minimum Common Supergraph

4 References :

Websites:

- https://dtai.cs.kuleuven.be/krr/files/bib/manuals/idp3-manual.pdf

Research Papers:

- Predicate Logic as a Modelling Language: The IDP System Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, Gerda Janssens, and Marc Denecker, https://arxiv.org/pdf/1401.6312.pdf
- Maurice Bruynooghe, Hendrik Blockeel, Bart Bogaerts, Broes De Cat, Stef De Pooter, Joachim Jansen, Anthony Labarre, Jan Ramon, Marc Denecker, and Sicco Verwer. Predicate logic as a modeling language: modeling and solving some machine learning and data mining problems with idp3. TPLP, FirstView:135, 6 2015.http://www.cs.ru.nl/ sicco/papers/bruynooghe13.pdf