DBLP API — A Shortest Path Algorithm

Appendix to the paper "DBLP — Some Lessons Learned" (June 9, 2009)

Michael Ley
Universität Trier, Informatik
D–54286 Trier
Germany
ley@uni-trier.de

The java program[1] documented in this appendix demonstrates the use of one of the DBLP XML services, which were described in section 5 of the paper "DBLP — Some Lessons Learned". It computes the shortest path between two DBLP authors in the coauthor graph. The software works like a little web crawler, it loads the information incrementally from the DBLP server.

The program is structured in two classes: The shortest path algorithm is a variant of the classic breasth-first algorithm, it is shown in figure 1. The class `Person` shown in figure 2 hides all DBLP specific implementation details from the algorithms itself.

The interaction beween both classes is very simple: If you have a `Person` object, you may ask it for its neighbors by applying the method `getCoauthors`, which returns an array of persons. You can attach a label to each person. A label is an integer. `setLabel` creates a label, `hasLabel` tests if a label exists, and `getLabel` reads a label of a person. The class method `resetAllLabels` deletes all labels from persons. It is obvious that you may generalize this to a simple interface to access any undirected graph (without weights) from your shortest path algorithm.

To run the algorithm, you have to create two persons and to construct a `CoauthorPath` object:

```
Person p1, p2;
p1 = Person.create("Jim Gray", "g/Gray:Jim");
p2 = Person.create(...);
CoauthorPath cp = new CoauthorPath(p1,p2);
Person path[] = cp.getPath();
```

After this you may print out the path.

Next we look at the class `Person` more closely. In the code snipplet above it is noticeable that we use the class method `create` to produce new `Person` objects. The class does not have a public constructor because it caches all objects in a class level `Map`. The `create` method first tests if there is already an object for the specified name in `personMap`. A new object is only created, if this test fails.

After creation a `Person` object only contains the person's name and the `urlpt`. The list of coauthors is only loaded on demand[2]. The boolean field `coauthorsLoaded` contains the required state information. If `getCoauthors` is called for the first time, `loadCoauthors` is activated to fetch the information from DBLP.

We use a SAX parser to read the XML file. Because the creation of a SAX parser object is an expensive task, the parser is reused. It is created in the static initialization block just above `loadCoauthors` at class load time. Additionally the `coauthorHandler` field is initialised. In `loadCoauthors` an `URL` object is constructed to provide an `InputStream` for the parser. During the parsing process the SAX parser calls the methods `startElement`, `endElement`, and `characters` of the local `CAConfigHandler`. We are only interested in `author` elements. The boolean `insideAuthor` is set `true` as soon as we recognize an opening `author` tag. `characters` collects the element contents in `Value`. In the method `endElement` a `Person` object is created. The coauthors are temporarily stored in `plist`. This `List` is converted into the final `coauthors` array after parsing has been completed.

To make the shortest path algorithm practicable over a slow internet connection, we have to minimize the number co-author lists to be loaded. Two known optimizations of the breadth-first algorithm proved to be essential for searches inside the huge connected component of the DBLP coauthor graph: (1) The search has to start from both persons, and (2) the algorithm should prefer the side with the lower number of persons to be visited next. The method `CoauthorPath.shortestPath` is a straightforward implementation of these ideas.

The algorithm labels persons `p1` with 1 and `p2` with -1. The direct neighbors of `p1` are set 2, the direct neighbors of `p2` are set -2 etc. The variables `now1` and `now2` contain the sets of persons who form the outer waves of the labeling processes. The main loop flips the side where to advance next depending on the size of these sets. During an expansion step the still unvisited persons are collected in the set `next`.

If `now1` or `now2` become empty, the persons are not connected. If the two waves hit each other, the method `tracing` is called to collect pathes from the meeting point to the starting points.

[1] An expanded version of this software is available from `http://dblp.uni-trier.de/xml/docu/`

[2] In the online version of the class the same mechanism is implemented for the list of publications of a person.

```
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;

public class CoauthorPath {
    private Person path[];

    public CoauthorPath(Person p1, Person p2)
        { shortestPath(p1,p2); }

    public Person[] getPath() { return path; }

    private void tracing(int position) {
        Person pNow, pNext;
        int direction, i, label;

        label = path[position].getLabel();
        direction = Integer.signum(label);
        label -= direction;
        while (label != 0) {
            pNow = path[position];
            Person ca[] = pNow.getCoauthors();
            for (i=0; i<ca.length; i++) {
                pNext = ca[i];
                if (!pNext.hasLabel())
                    continue;
                if (pNext.getLabel() == label) {
                    position -= direction;
                    label -= direction;
                    path[position] = pNext;
                    break;
    }   }   }   }

    private void shortestPath(Person p1, Person p2) {
        Collection<Person>  h,
          now1 = new HashSet<Person>(),
          now2 = new HashSet<Person>(),
          next = new HashSet<Person>();
        int direction, label, n;

        Person.resetAllLabels();
        if (p1 == null || p2 == null)
            return;
        if (p1 == p2) {
            p1.setLabel(1);
            path = new Person[1];
            path[0] = p1;
            return;
        }

        p1.setLabel( 1); now1.add(p1);
        p2.setLabel(-1); now2.add(p2);

        while (true) {
            if (now1.isEmpty() || now2.isEmpty())
                return;

            if (now2.size() < now1.size()) {
                h = now1; now1 = now2; now2 = h;
            }

            Iterator<Person> nowI = now1.iterator();
            while (nowI.hasNext()) {
                Person pnow = nowI.next();
                label = pnow.getLabel();
                direction = Integer.signum(label);
                Person neighbours[] = pnow.getCoauthors();
                int i;
                for (i=0; i<neighbours.length; i++) {
                    Person px = neighbours[i];
                    if (px.hasLabel()) {
                        if (Integer.signum(px.getLabel())
                                ==direction)
                            continue;
                        if (direction < 0) {
                            Person ph;
                            ph = px; px = pnow; pnow = ph;
                        }
                        // pnow has a positive label,
                        // px a negative
                        n = pnow.getLabel() - px.getLabel();
                        path = new Person[n];
                        path[pnow.getLabel()-1] = pnow;
                        path[n+px.getLabel()] = px;
                        tracing(pnow.getLabel()-1);
                        tracing(n+px.getLabel());
                        return;
                    }
                    px.setLabel(label+direction);
                    next.add(px);
                }
            }
            now1.clear(); h = now1; now1 = next; next = h;
        }
    }
}
```

Figure 1: A shortest path algorithm

```
import ...;

public class Person {
    private static Map<String, Person> personMap =
            new HashMap<String, Person>();
    private String name;
    private String urlpt;

    private Person(String name, String urlpt) {
        this.name = name;
        this.urlpt = urlpt;
        personMap.put(name, this);
        coauthorsLoaded = false;
        labelvalid = false;
    }

    static public Person create(String name, String urlpt) {
        Person p;
        p = searchPerson(name);
        if (p == null)
            p = new Person(name, urlpt);
        return p;
    }

    static public Person searchPerson(String name) {
        return personMap.get(name);
    }

    private boolean coauthorsLoaded;
    private Person coauthors[];

    static private SAXParser coauthorParser;
    static private CAConfigHandler coauthorHandler;
    static private List<Person> plist
            = new ArrayList<Person>();

    static private class CAConfigHandler
                extends DefaultHandler {
        private String Value, urlpt;
        private boolean insideAuthor;

        public void startElement(String namespaceURI,
                String localName, String rawName,
                Attributes atts) throws SAXException {
            if (insideAuthor = rawName.equals("author")) {
                Value = "";
                urlpt = atts.getValue("urlpt");
        }   }

        public void endElement(String namespaceURI,
                    String localName, String rawName)
                        throws SAXException {
            if (rawName.equals("author") &&
                    Value.length() > 0) {
                plist.add(create(Value, urlpt));
        }   }

        public void characters(char[] ch, int start, int length)
                throws SAXException {
            if (insideAuthor)
                Value += new String(ch, start, length);
        }

        public void warning(SAXParseException e)
                throws SAXException { ... }
        public void error(SAXParseException e)
                throws SAXException { ... }
        public void fatalError(SAXParseException e)
                throws SAXException { ... }
    }

    static {
        try { coauthorParser = SAXParserFactory.
              newInstance().newSAXParser();
            coauthorHandler = new CAConfigHandler();
            coauthorParser.getXMLReader().setFeature(
             "http://xml.org/sax/features/validation",
             false);
        } catch (ParserConfigurationException e) { ...
        } catch (SAXException e) { ... }
    }

    private void loadCoauthors() {
        if (coauthorsLoaded)
            return;
        plist.clear();
        try { URL u = new URL(
            "http://dblp.uni-trier.de/rec/pers/"
             +urlpt+"/xc");
            coauthorParser.parse(u.openStream(),
                        coauthorHandler);
        } catch (IOException e) { ...
        } catch (SAXException e) { ... }
        coauthors = new Person[plist.size()];
        coauthors = plist.toArray(coauthors);
        coauthorsLoaded = true;
    }

    public Person[] getCoauthors() {
        if (!coauthorsLoaded) { loadCoauthors(); }
        return coauthors;
    }

    private int label;
    private boolean labelvalid;

    public int getLabel() { return (!labelvalid)?0:label; }
    public void resetLabel() { labelvalid = false; }
    public boolean hasLabel() { return labelvalid; }

    public void setLabel(int label) {
        this.label = label;
        labelvalid = true;
    }

    static public void resetAllLabels() {
        Iterator<Person> i = personMap.values().iterator();
        while (i.hasNext()) {
            Person p = i.next();
            p.labelvalid = false;
            p.label = 0;
    }   }

    public String toString() { return name; }
}
```

Figure 2: The Class "Person"