Relational PC User's Guide

Relational PC User's Guide

Published July 09, 2010

This effort is supported by AFRL and MIT Lincoln Laboratory under contract numbers FA8750-09-2-0187 and 7000032881. The U.S. Government is authorized to re- produce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied, of AFRL, MIT or the U.S. Government.

General inquiries regarding Relational PC should be directed to:

Knowledge Discovery Laboratory
c/o Professor David Jensen, Director
Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003-9264

or

kdlsupport@kdl.cs.umass.edu

Table of Contents

Chapter 1. Installation

Requirements

The Relational PC package (RPC) is an experimental prototype and has not been designed to support multiple platforms or alternate versions of the required components. Currently, Relational PC runs under Mac OS X 10.5. Due to lack of support for the SWI-Prolog JPL libraries, Mac OS X 10.6 is explicitly not supported, and it is unlikely that versions of OS X earlier than 10.4 will support all the applications necessary to run Relational PC. Although the package may run on other selected UNIX platforms, this has not been tested and is not supported. The instructions for installing and running RPC assume that you are working under Mac OS X 10.5.

The RPC package requires the following components:

- Java 5.0 or higher
- SWI-Prolog 5.6.64 or higher
- R 2.10 or higher
- PostgreSQL 8.4 or higher
- rJava 0.7-0 (Java–R interface library)

Optionally, you may also want to install Graphviz or other graph visualization software capable of displaying dot (hierarchical) drawings.

Installing Relational PC

Follow the steps below to make sure that you have the necessary components for running RPC. You must have administrator privileges to install some of the required components.

Installing Relational PC:

1. Verify that you are using Java SE 5.0 or higher (i.e., version 1.5 or higher). You can check your Java version with the command

   ```
   > java -version
   ```
2. Download and install SWI-Prolog 5.6.64 or higher from www.swi-prolog.org.

   Add the location of the swipl executable to `PATH`. The default location for the swipl executable on Mac OS X is in `/opt/local/bin`.
3. Download and install R 2.10 or higher from www.r-project.org.
4. Download and install PostgreSQL 8.4 or higher from www.postgresql.org.

   Make sure that you record the password you assign to the PostgreSQL admin user.
5. Create a user with your username inside PostgreSQL.
   a. In Finder, navigate to the PostgreSQL directory. For a default Mac OS X PostgreSQL installation, this is likely to be "`/Applications/PostgreSQL` *`version`*" where *`version`* is the version of PostgreSQL.
   b. Double click the pgAdmin application to open the application.
   c. In the **Object browser** pane, double click the PostgreSQL server name.
   d. When prompted, enter the PostgreSQL admin user password.
   e. From the **Edit** menu, choose **New Object**, then choose **New Login Role**.
   f. In the **Role name** box, enter your login name for your local computer.
   g. In the **Password** box, enter a password for this user. The password need not be the same as your login password.
   h. Retype the password in the **Password (again)** box.

   i. Click **Role privileges** and select all options.
   j. Click **OK**.

6. Create a directory to hold the PostgreSQL log files. You will probably need to use `sudo` to make changes to the PostgreSQL installation.

   a. Change to your PostgreSQL directory. For Mac OS X this is typically `/Library/PostgreSQL/`*`version`*.

   ```
   > cd /Library/PostgreSQL/version
   ```

   b. Create the `logs` subdirectory.

   ```
   > sudo mkdir logs
   ```

   Enter your usual user password when prompted.

   c. Change ownership of the new directory to `postgres`.

   ```
   > sudo chown postgres logs
   ```

7. Configure PostgreSQL to trust local connections.

   a. Switch to the “postgres” user identity

   ```
   > sudo su postgres
   ```

   b. Edit the file `/Library/PostgreSQL/`*`version`*`/data/pg_hba.conf`, changing the local and IPv4 connection methods to `trust`.

   ```
   # "local" is for Unix domain socket connections only
   local   all         all                               trust
   # IPv4 local connections:
   host    all         all         127.0.0.1/32          trust
   # IPv6 local connections:
   host    all         all         ::1/128               md5
   ```

   c. Save the edited file and stay logged in as the “postgres” user.

8. Manually restart the PostgreSQL server, allowing it to use as much working memory as possible.

   a. Still logged in as the “postgres” user, stop the PostgreSQL server.

   ```
   > cd /Library/PostgreSQL/version/bin
   > ./pg_ctl stop -D ../data
   ```

   b. Restart the server, providing additional memory.

   ```
   > ./postmaster -S 1000000 -D ../data/ > ../logs/logfile 2>&1 &
   ```

   c. Log out as the “postgres” user.

   ```
   > exit
   ```

   If you stop the PostgreSQL server, you must restart it using the same command in order to enable the increase in working memory. Alternately, you can edit `postgresql.conf` to reset the available working memory persistently.

   a. Logged in as the “postgres” user, edit the file `/Library/PostgreSQL/`*`version`*`/data/postgresql.conf`, uncommenting and increasing the value of `work_mem`.

   ```
   work_mem = 1024MB                        # min 64kB
   ```

   The 1024MB allocated above is shown as an example value. Choose a size for working memory suitable for your machine’s architecture and your data needs.

   b. Stop and restart the PostgreSQL server for this setting to take effect.

   ```
   > cd /Library/PostgreSQL/version/bin
   > ./pg_ctl stop -D ../data
   ```

```
> ./postmaster -D ../data/ > ../logs/logfile 2>&1 &
```

9. Download and unpack the RPC distribution available from kdl.cs.umass.edu/causality. Although you may install the Relational PC package wherever you wish, this document assumes installation in the `rpc` directory at the top level in your user directory, i.e., `/Users/`*`username`*`/rpc/`.
10. Copy the Java–Prolog interface dynamic library (`libjpl.dylib`) from the SWI-Prolog installation to the `lib` directory of your RPC installation.
    a. Find your local SWI-Prolog installation. For a default installation on Mac OS X your local SWI-Prolog directory is likely to be `/opt/local/lib/swipl-`*`version`*. If this is not the case, examine the symbolic link target for the swipl executable.

       For example, the link might point to the `/opt/local/lib/swipl-5.10.0/bin/i386-darwin9.8.0` directory. Your local SWI-Prolog installation would therefore be in `/opt/local/lib/swipl-5.10.0`.
    b. Navigate to your local SWI-Prolog installation directory.

       ```
       > cd /opt/local/lib/swipl-version
       ```
    c. Go to the `lib` subdirectory for your specific architecture. For example,

       ```
       > cd lib/i386-darwin9.8.0
       ```
    d. Copy the file `libjpl.dylib` to the `lib` directory of your RPC installation.

       ```
       > cp libjpl.dylib ~/rpc/lib/
       ```
    e. Change to the `lib` directory of your RPC installation and create a symbolic link from the SWI dynamic library to the corresponding `jnilib` file.

       ```
       > cd ~/rpc/lib
       > ln -s libjpl.dylib libjpl.jnilib
       ```
11. Create your machine-dependent Java–R interface (JRI) library for RPC.
    a. Download and uncompress rJava 0.7-0 from cran.r-project.org/src/contrib/Archive/rJava. You must use version 0.7-0; both newer and older versions do not support the needed functionality.
    b. Change to the `jri` subdirectory of your rJava installation.
    c. Follow the instructions in the `README` file in this directory to compile the library. You may need to execute the compilation commands with sudo.
    d. Copy `libjri.jnilib` into the `lib` directory of your RPC installation.
12. Change to the `build` subdirectory of your local RPC installation and compile RPC using the ant build tool.

    ```
    > cd ~/rpc/build
    > ant
    ```
13. Define the required environment variables.

    Edit or create `/etc/launchd.conf` to include the following lines:

    ```
    setenv RPC_HOME /Users/username/rpc
    setenv R_HOME /Library/Frameworks/R.framework/Resources
    ```

    where *`username`* is your computer username. Substitute the appropriate path if you have installed RPC in a different location.

    You must reboot your computer for these definitions to take effect.

To test your installation, run the example `runRPC.py` script included in the `$RPC_HOME/example/script/` directory. This script generates a small example database from a known causal model and runs the RPC algorithm on that data.

```
> cd $RPC_HOME
> bin/script.sh example/scripts/runRPC.py testdb example/schemas/datagen-schema.pl
```

Chapter 2. Running Relational PC

Scripting Overview

RPC is run via Jython scripts. Using Jython permits access to the full RPC Java class library. Source code files for the scripts discussed in this section are available in `$RPC_HOME/example/scripts`.

Importing RPC packages

You only need to import a class or package when you refer to a class by name, such as when instantiating a class object, using a static method, or referring to one of its class variables.

Executing scripts

To execute an RPC script, call `$RPC_HOME/bin/script.sh` from a terminal window, passing the required arguments on the command line. You must provide the name of the Jython script; other aruments are optional and depend on the requirements of your Jython script.

To execute the example scripts included in this release, use

```
> $RPC_HOME/bin/script.sh script-file db-name schema-file
```

where

- *`db-name`* is the name of the database, and
- *`script-file`* and *`schema-file`* must be relative paths within the RPC installation tree or fully qualified paths to the corresponding files.

Controlling logging

The Relational PC package uses log4j to control the level of logging information displayed. To control the logging options, RPC looks for a configuration file named `rcp.lcf` in the directory from which you execute the `script.sh` command. The default set up of RPC provides no logging configuration file and logs only the major milestones and output of the algorithm.

You can include additional details in the logging trace by specifying a configuration file for log4j. An example configuration file that provides detailed traces of the algorithm’s actions, including queries and statistical tests, is included in `$RPC_HOME/example/config/rcp.lcf`. Copy the example configuration file to `$RPC_HOME` to use its settings. Or create a new configuration file, name it `rcp.lcf`, and place it in `$RPC_HOME` to define your own logging options.

Running RPC on Generated Data

The `runRPC.py` script illustrates how to run RPC on generated data. The generated data used in this example is described by the schema file `$RPC_HOME/example/schemas/datagen-schema.pl`. This schema file is discussed in detail in Chapter 3, later in this document.

Import needed classes.

```
from rpc.datagen import DataGenerator
from rpc.datagen import CausalModelGenerator
from rpc.dataretrieval import Database
from rpc.model.util import ModelSupport
from rpc.model import RelationalPC
from rpc.model.scoring import ModelScoring
```

The database name and path to the schema file are passed in on the command line. The script loads the Prolog code module and schema file for the target database.

```
dbName = ckd.args[0]
schemaName = ckd.args[1]

ckd.loadRPC()
ckd.loadSchema(schemaName)
```

Set parameter values.

- `hopThreshold` determines he number of “hops” from a starting entity or relationship that the algorithm can travel when creating units.
- `maxParents` is used by data generation to set a limit on the number of parents a node may have.
- `rpcDepth` determines the maximum size of the conditioning set.
- `numDependencies` is used by data generation to set the number of dependencies randomly created in the true model of the generated data.
- `sampleSize` is provided as a convenience for this script; it is used in determining the size of the tables for each entity.

```
hopThreshold = 2
maxParents = 2
rpcDepth = 2
numDependencies = 10
sampleSize = 800
```

Generate a random (known) causal model structure for use in data generation. Instantiate the causal model and remember the true models for later use (trueModel). Optionally, write out the true model for visualization.

```
cg = CausalModelGenerator(numDependencies, hopThreshold, maxParents)
trueModel = cg.getModel()
#trueModel.getDotFile("trueModel_dep" + str(numDependencies) + ".dot", 1)
```

Parameterize the dependencies in the causal model. We first parameterize the model to specify the average cardinalities and conditional probability distributions of attributes for related entities. For example, the code below specifies that there will be an average of two `b` entities linked to each `a` entity. Attribute values are assigned randomly within the constraints imposed by the model’s structure.

```
p = cg.parameterize({"ab":2.0, "da":1.0, "bc":2.0})
```

Generate data from the known causal model structure and parameters. In this code, we specify that the generated data will include twice as many `b` entities and half as many `c` entities as there are `a` and `d` entities.

```
dg = DataGenerator(dbName)
dg.generate(cg, p, {"a":sampleSize, "b":2*sampleSize, "c":sampleSize/2,
        "d":sampleSize})
dg.close()
```

Open the newly generated database and set the hop threshold.

```
Database.open(dbName)
ckd.setHopThreshold(hopThreshold)
```

Identify potential units in the generated data. Units are created by combining two “paths,” each of which corresponds to a base entity or relationship with the possible addition of entities and relationships up to two hops (`hopThreshold`) away from that base item. Paths are paired with other paths sharing the same base entity or relationship to create a unit consisting of an outcome path and a treatment path. The algorithm converts the outcome path into a singleton and combines the rest of the unit structure into the treatment path.

For example, the pair of paths [a ab b].y and [a da d].w can be combined because they share a common base item (the entity a), yielding an outcome path of d.w and a treatment path of [d da a ab b].y.

```
allUnits = ckd.getUnits()
uniqueUnits = ckd.getUniqueUnits()
```

Initialize model support data structure.

```
ckd.modelSupport = ModelSupport(allUnits, uniqueUnits, hopThreshold)
```

Set the significance level (alpha) and strength of effect. The significance threshold (alpha) is Bonferroni-adjusted due to the multiple significance tests run. RPC ignores any possible dependencies with a strength of effect less than that specified.

```
rpc = RelationalPC(ckd.modelSupport)
rpc.setSignficanceThresholdAdjust(0.01)
rpc.setStrengthOfEffectThreshold(0.1)
```

Phase I: Skeleton identification. Optionally, write the learned (before edge orientation) model to a file for visualization.

```
rpc.identifySkeleton(rpcDepth)
learnedModel = rpc.getModel()
#learnedModel.getDotFile("learnedModelPhaseI_dep" + str(numDependencies)
        + "_ss" + str(sampleSize) + ".dot", 1)
```

Score the model. At this point, the learned model is still undirected. Determine how many edges can be trivially oriented at this point (we need the `errorCts` to get this value).

```
errorCts = ModelScoring.getErrorCounts(trueModel, learnedModel)
trivialOrient = list(errorCts)[0]
```

Phase II: Edge orientation.

```
rpc.orientEdges()
```

Print out the resulting constraints and dependencies. Optionally, save the learned model (after edge orientation) to a file for visualization.

```
print "Independence constraints:"
for constraint in rpc.getConstraints():
        print "\t" + constraint

print "Dependencies:"
for dependence in rpc.getDependencies():
        print "\t" + str(dependence)

learnedModel = rpc.getModel()
#learnedModel.getDotFile("learnedModelPhaseII_dep" + str(numDependencies)
        + "_ss" + str(sampleSize) + ".dot", 1)
```

Score the model.

```
errorCts = ModelScoring.getErrorCounts(trueModel, learnedModel)

sprecision = ModelScoring.getSPrecision(errorCts)
srecall = ModelScoring.getSRecall(errorCts)
cprecision = ModelScoring.getCPrecision(errorCts)
crecall = ModelScoring.getCRecall(errorCts)

ruleFreqs = ModelScoring.getEdgeRuleFrequencies()
cd = ruleFreqs.get("CD") if ruleFreqs.containsKey("CD") else 0
rem = ruleFreqs.get("REM") if ruleFreqs.containsKey("REM") else 0
knc = ruleFreqs.get("KNC") if ruleFreqs.containsKey("KNC") else 0
```

```
ca = ruleFreqs.get("CA") if ruleFreqs.containsKey("CA") else 0

oracleCRecall = ModelScoring.getOracleCRecall(ckd.modelSupport, trueModel)
```

Print evaluation metrics. The “S” values refer to the undirected skeleton, and the “C” values refer to the compelled (directed) model. The edge rule frequencies tell us how many times each edge-orientation rule was used. Relational PC uses the following edge-orientation rules:

`cd`	collider detection
`rem`	restricted existence models
`knc`	known non-collider
`ca`	cycle avoidance

`OracleCRecall` tells us how many of the edges RPC would orient correctly if given the true undirected skeleton.

```
print "SPrecision:", sprecision
print "SRecall:", srecall
print "CPrecision:", cprecision
print "CRecall:", crecall
print "Triv:", trivialOrient
print "Edge Rule Frequencies:", "cd:", cd, "rem:", rem, "knc:", knc, "ca:", ca
print "Oracle CRecall:", oracleCRecall
```

Close the database connection.

```
Database.close()
```

Running runRPC.py

To execute the `runRPC.py` script, change to the root of your local RPC installation and type the following command:

```
> cd $RPC_HOME
> bin/script.sh example/scripts/runRPC.py testdb example/schemas/datagen-schema.pl
```

Interpreting RPC results

The output from running `runRPC.py` contains two important sections:

- a list of independence constraints that describe all possible treatment and outcome pairs
- a list of dependencies identifying the units for which the algorithm could not conditionally separate the treatment and outcome

For example, the independence constraint

```
Path [c, bc, b], AttrVar: b.y _||_ Path: [c], AttrVar: c.z | []
```

indicates that the outcome variable (the `z` attribute on the `c` entity) is conditionally independent of the treatment (the `y` attribute on the `b` entity related to `c` through the `bc` relationship), given nothing (the final “`[]`” in the statement).

The dependency

```
(Base: b, Treatment Path <Path: [b, ab, a], AttrVar: a.x>,
          Outcome Path <Path: [b], AttrVar: b.y>)
```

Indicates that RPC could not conditionally separate the treatment variable (the `x` attribute on the `a` entity in the unit `[b, ab, a]`) from the outcome variable (the `y` attribute on the `b` entity).

Running RPC on Existing Data

The `runRPCUsingDatabase.py` script illustrates how to run RPC on existing data.

Import needed classes.

```
from rpc.dataretrieval import Database
from rpc.model.util import ModelSupport
from rpc.model import RelationalPC
```

Set parameter values.

- `rpcDepth` determines the maximum size of the conditioning set.
- `hopThreshold` determines he number of "hops" from a starting entity or relationship that the algorithm can travel when creating units.

```
rpcDepth = 2
hop_threshold = 2
```

The database name and path to the schema file are passed in on the command line. The script loads the Prolog code module and schema file for the target database. Set the hop threshold.

```
Database.open(ckd.args[0])
ckd.loadRPC()
ckd.loadSchema(ckd.args[1])

ckd.setHopThreshold(hop_threshold)
```

Identify potential units in the database. Combine paths having the same base entity or relationship to create outcome and treatment paths.

```
allUnits = ckd.getUnits()
uniqueUnits = ckd.getUniqueUnits()
```

Initialize model support data structure.

```
ckd.modelSupport = ModelSupport(allUnits, uniqueUnits, ckd.getHopThreshold())
```

Set the significance level (alpha) and strength of effect.

```
rpc = RelationalPC(ckd.modelSupport)
rpc.setSignficanceThresholdAdjust(0.01)
rpc.setStrengthOfEffectThreshold(0.1)
```

Phase I: Skeleton identification. Optionally, write the learned (before edge orientation) model to a file for visualization.

```
rpc.identifySkeleton(rpcDepth)
learnedModel = rpc.getModel()
#learnedModel.getDotFile("LearnedModelPhaseI.dot", 1)
```

Phase II: Skeleton identification. Optionally, write the learned (after edge orientation) model to a file for visualization.

```
rpc.orientEdges()
learnedModel = rpc.getModel()
#learnedModel.getDotFile("LearnedModelPhaseII.dot", 1)
```

Print out the resulting constraints and dependencies.

```
print "Independence constraints:"
for constraint in rpc.getConstraints():
        print "\t" + constraint

print "Dependencies:"
for dependence in rpc.getDependencies():
        print "\t" + str(dependence)
```

Close the database connection.

```
Database.close()
```

Running runRPCUsingDatabase.py

To execute the `runRPCUsingDatabase.py` script, change to the root of your local RPC installation and type the following command:

```
> cd $RPC_HOME
> bin/script.sh example/scripts/runRPC.py db-name schema-file
```

where *`db-name`* is the name of the PostgreSQL database being served and *`schema-file`* is the path to the file defining the schema for this database..

Chapter 3. Data for Relational PC

You can use your own data or generated data for RPC. This chapter describes the required format for imported data and database schema files. See the description of the `runPRC.py` script in “Running RPC on Generated Data” for details on generating data to use with RPC.

Data Requirements

To use your own data with RPC you must

- Make sure that the data obeys the required format
- Provide a schema file for the data

The format for the schema file is described below. The RPC algorithm ignores any tables that are not included in the schema file.

Data requirements for use with RPC are fairly straightforward:

- Every entity and relationship must have a corresponding table in the database.
- Every table must have a primary key.
- Every relationship table must contain foreign keys to the corresponding entity tables.

By convention, key names default to

```
name_id
```

where *`name`* is the name of the entity or relationship defined by that table. You can define different primary and foreign keys using the `primaryKey` and `foreignKey` statements in your schema file.

For example:

```
primaryKey(person_id, people).
foreignKey(resides_id, livesAt, people).
```

defines the primary key `person_id` for the “people” table and defines the foreign key `resides_id` for references to the “people” table from the “livesAt” table. See `$RPC_HOME/test/test-schema.pl` for additional examples of defining primary and foreign keys in the schema file.

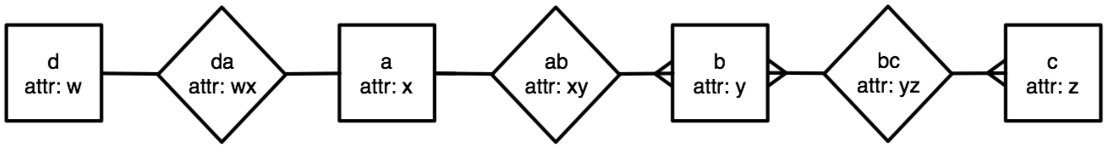
Database Schema

The Relational PC package uses a Prolog file to define the associated database schema. You must define four schema components:

- entities
- relationships
- attributes on entities
- attributes on relationships

RPC treats all numerical values as continuous data, defining bins to create categorical data for χ^2 statistical operations. If your data contains numerical values that should be treated as categorical data, you can specify this in the schema file, as well.

The code fragments in this section define the schema used for the data generated for the example `runRPC.py` script:



Entities

The example schema contains four entities, labeled a through d. Each entity is defined by a line in the schema file of the form

`entity(`*`label`*`).`

The entities in the example schema are defined by the following lines:

```
entity(a).
entity(b).
entity(c).
entity(d).
```

Relationships

The example schema contains three relationships, labeled for the entities they connect. Each relationship is defined by a line in the schema file of the form

`relationship(`*`label`*`).`

The relationships in the example schema are defined by the following lines:

```
relationship(ab).
relationship(bc).
relationship(da).
```

Note that relationships in an RPC schema are undirected.

You must also specify the cardinality of each relationship. Relationships can be one-to-one, one-to-many, or many-to-many. The cardinality of each relationship “end point” is defined separately, using a line of the form

`cardinality(`*`relationship-name`*`, `*`end-point`*`, `*`value`*`).`

where

`value` is one of “`one`” or “`many`”.

The cardinality of the relationships in the example schema are defined by the following lines:

```
cardinality(ab, a, one).
cardinality(ab, b, many).
cardinality(bc, b, many).
cardinality(bc, c, many).
cardinality(da, d, one).
cardinality(da, a, one).
```

These lines specify that ab defines a one-to-many relationship linking a and b, bc defines a many-to-many relationship linking b and c, and da defines a one-to-one relationship linking d and a.

Attributes on entities

The example schema defines one attribute for each entity, labeled w through z. Each entity attribute is defined by a line in the schema file of the form

`attr(`*`attr-name`*`, `*`entity`*`).`

The entity attributes in the example schema are defined by the following lines:

```
attr(x, a).
attr(y, b).
attr(z, c).
attr(w, d).
```

Attributes on relationships

The RPC algorithm is able to use information from attributes on both entities and relationships. The example schema defines one attribute for each relationship, labeled for the attributes on the entities that they connect. Each relationship attribute is defined by a line in the schema file of the form. Relationship attributes are defined the same way as entity attributes:

`attr(`*`attr-name, relationship`*`).`

The relationship attributes in the example schema are defined by the following lines:

```
attr(w, d).
attr(wx, da).
attr(x, a).
attr(xy, ab).
attr(y, b).
attr(yz, bc).
attr(z, c).
```

Categorical data

RPC currently requires all attribute values be numeric. To use categorical values, you must convert non-numeric categorical values to integers and specify that they are to be treated categorically, as described below.

By default, RPC treats all numerical data as continuous, binning values to create categorical data for use by χ^2 statistical operations. If your database contains categorical numeric attribute values that should not be binned, specify this in the schema file, as shown below:

`categorical(`*`attr-name`*`).`