PReMo - User's Manual

Dominik Wojtczak

Contents

1 Introduction

This document describes the tool PReMo(Probabilistic Recursive Models analyzer) and illustrate it on some simple examples. PReMo(read as *primo*) is capable of analyzing Recursive Markov Chains(RMCs), Recursive Simple Stochastic Games(RSSGs) and Stochastic Context Free Grammars(SCFGs). RMCs are a natural model for recursive imperative programs with probabilistic transitions. Probability comes either from an explicit randomization like in the Quicksort algorithm or by abstracting some aspects of a program. A very tightly related model to this one is Probabilistic Pushdown Systems (pPDSs). There exist polynomial time algorithms translating one model into the other.

The theory behind RMCs and pPDSs was investigated and published in a whole series of papers [2, 4, 1]. There are many interesting question we could ask about a given Recursive Markov Chain such as computation of termination probability, expected time of termination or its variance. Algorithms were given in [2, 1], that use results from the Existential Theory of Reals, allowing us to decide in PSPACE whether those values are greater than a given rational value. On top of that, in [2], a reduction was established from the well known Square Root Sum problem to the problem of computing the termination probabilities for RMCs. This means that finding an algorithm with substantially better complexity than PSPACE, would solve a long standing open problem. On the other hand we have efficient numerical approximation algorithms described in [2] for computing all these values by computing the least fixed point of a specific nonlinear system of equations, eg. using Newton's method. Related to RMCs, but placed in the game theoretic setting, are Recursive Simple Stochastic Games[3] where we allow some of the nodes to be controlled by players. The player called *maximizer* tries to maximize eg. the termination probability, while *minimizer* tries to minimize it.

The program has four input modes: *RMC* for typing as an input a source of Recursive Markov Chain, *RSSG* for modeling Recursive Markov Decision Processes and Recursive Simple Stochastic Games, *SCFG* for Stochastic Context Free Grammar and *Equations* where the user can type any set of recursive equations with standard functions and the system will try to find the fixed point if one exists starting at all zero vector.

2 Input languages

2.1 Recursive Markov Chains

We can see an example Recursive Markov Chain in the Fig. 1.

```
A(2,2);
B(1,2);

A {
    L1(A);
    L2(B);
    entry 0:
        0.5: goto L3; 0.5: call L2(0);
    entry 1:
        0.3: call L1(0); 0.7: call L1(1);
    L1 {
        exit 0:
            0.8: goto L3; 0.2: call L2(0);
        exit 1:
            1.0: return 1;
    }
    L2 {
        exit 0:
            1.0: goto L3;
        exit 1:
            0.5: return 0; 0.5: return 1;
    }
    L3 {
        0.5: return 0; 0.5: return 1;
    }
}

B {
    L1(A);
    entry 0:
        0.3: call L1(0); 0.3: call L1(1); 0.4: goto L2;
    L1 {
        exit 0:
            0.5: return 0; 0.5: return 1;
        exit 1:
            0.5: return 0; 0.5: goto L2;
    }
    L2 {
        0.4: goto L2; 0.6: return 0;
    }
}
```

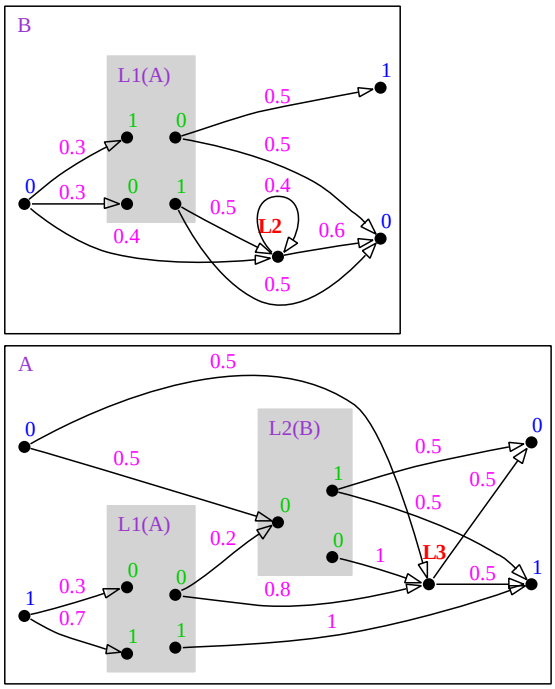

Figure 1: Source code of an example Recursive Markov Chain and its underlying transition graph

The PReMo syntax for defining RMCs is as follows: first we declare all components (procedures), using two numbers as parameters which denote the number of entries and exits; for instance in the example showed on fig. 1 `B(1,2);` declares "Component B has 1 entry and 2 exits". Next, we define the components. Each component definition starts with a declaration of all the boxes contained in the component, together with the component it maps to, e.g., `L2(B);` declares a box named L2 that is mapped to component B. Next for all entries, internal nodes, and exits of the boxes, we specify a list of transitions available from that control state. A single transition is a probability followed by a `goto`, `call` or `return` instruction separated by a colon. We use `goto` instruction when the transition leads to an internal node of this component. An instruction `call name_of_the_box(entry_number)` is used when we want to call the component that is mapped to the box labeled name_of_the_box with a parameter entry_number. Finally we use `return`

`value;` when we want to exit this component and return value value.

A formal grammar of the input language for RMCs is specified in 4.1.

2.2 Recursive Markov Decision Processes and Recursive Simple Stochastic Gamees

We can see an example source code of a RSSG with its transition graph on the fig. 2. The specification of an RSSG is the same as for an RMC except that any entries, internal nodes or box-exits can be preceded by the name of the player(min or max) in square brackets eg. `[max] L1 {...}` makes the internal node `L1` be controlled by the *maximizer*. On the transition graph drawing nodes controlled by the *maximizer* are painted red, while the nodes controlled by the *minimizer* are painted blue and the random nodes are black as for the RMCs.

```
A(2,2);
B(1,2);

A {
    L1(A);
    L2(B);
    [min]entry 0:
        goto L3; call L2(0);
    entry 1:
        0.3: call L1(0); 0.7: call L1(1);
    L1 {
        exit 0:
            0.8: goto L3; 0.2: call L2(0);
        exit 1:
            1.0: return 1;
    }
    L2 {
        exit 0:
            1.0: goto L3;
        [max]exit 1:
            return 0; return 1;
    }
    L3 {
        0.5: return 0; 0.5: return 1;
    }
}

B {
    L1(A);
    entry 0:
        0.3: call L1(0); 0.3: call L1(1); 0.4: goto L2;
    L1 {
        [min]exit 0:
            return 0; return 1;
        exit 1:
            0.5: return 0; 0.5: goto L2;
    }
    [max]L2 {
        goto L2; return 0;
    }
}
```

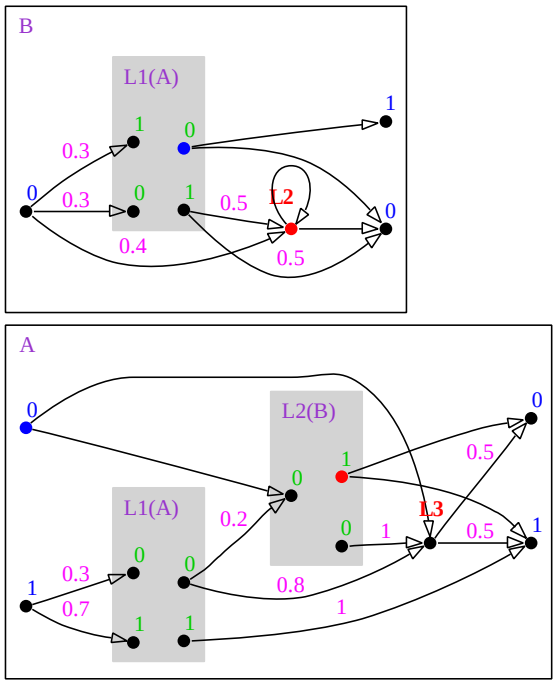

Figure 2: Source code of an example Recursive Simple Stochastic Game and its underlying transition graph

A formal grammar of the input language for Recursive Markov Decision Processes and Recursive Simple Stochastic Gamees is defined in 4.2.

2.3 Stochastic Context Free Grammars

To give Stochastic Context Free Grammar as an input the user specify a set of productions, one on each line. Each production specifies firstly the probability of this productions, then the non-terminal that we will be replacing, a symbol `->` followed by any sequence of terminals and non-terminals. The user does not have to specify which symbols are terminals as the program assumes that any symbol without a production attached to it is a terminal. For an example SCFG shown on the fig. 3, the only terminal is D and the non-terminals are A, B and C.

```
0.5 A -> B C D
0.5 A -> A A
0.7 B -> C
0.3 B -> A
0.4 C -> B
0.6 C -> D
```

Figure 3: Source code of an example Stochastic Context Free Grammar

A formal grammar of the input language for Stochastic Context Free Grammars is defined in 4.3.

2.4 Arbitrary equations systems

Here we define a list of equations separated by a newline character. Each equation defines a recursive expression that this variable has to fulfill eg. `x = cos(y) + x` defines an equation for the variable x and its dependence on the variables y and x. A formal grammar in short could look like this:

$$\langle Equation \rangle ::= \langle Variable \rangle = \langle Expression \rangle$$

I am not going to go in details how expressions in our grammar looks like, see `http://www.singsurf.org/djep/html/grammar.html` for more details what expressions are allowed. List of supported functions is available here: `http://www.singsurf.org/djep/html/grammar.html`. On top of that we added functions max and min with arbitrary number of arguments. Obviously if such function occurs in our equation system, Newton method won't work as it won't be able to differentiate them.

3 Using PReMo in practice

3.1 Creating and opening a new source file

First, we create a new or open an existing file using appropriate model from the File menu (fig. 4. We have a choice of four models RMC, RSSG, SCFG and Equation system described in details in section 2.

PReMo has a source code editor that support syntax highlighting, auto-indentation and if a parsing error occurred, jumping and selecting that line. The user can save the code or graph source, generated equations and found solution with performance analysis to an external file.

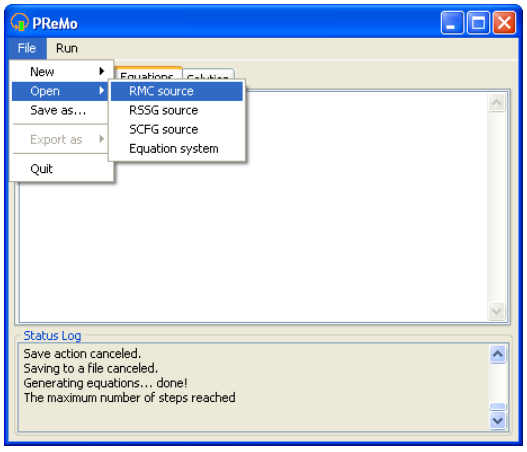

Figure 4: File Menu

In the case we have chosen to open a file, a file selection dialog appears with an extension filter depending on what type of a model we have chosen to open.

After typing a new model or opening a file we might in an example RMC as seen on fig. 5.

3.2 Parsing

When our source code is ready, we need to parse it in order to create a model. Based on that model the system will be able to draw a transition graph of our model (only for RMCs and RSSGs) or create an equation system (for termination probability or expected termination time) and then solve it using one of many numerical algorithms available in the program.

In order to parse the file we choose from the *Run* menu option *Parse*. If the parsing was successful, in the *Status Log* display we will see appropriate

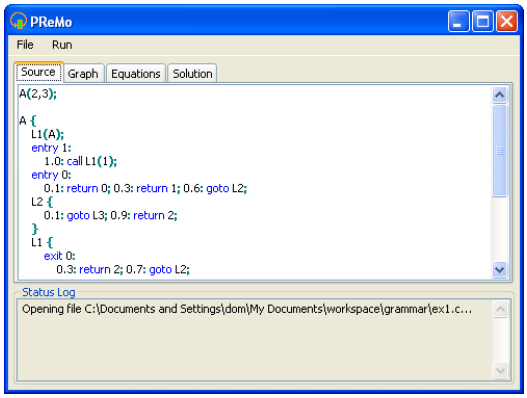

Figure 5: Example source editing

message with a short summary of the model. If there was an error the selected line where the parser failed will become highlighted(although it isn't 100% accurate, because the parser might have parsed few buggy lines before noticing an error) and the error description will be displayed in the *Status Log* (fig. 6).

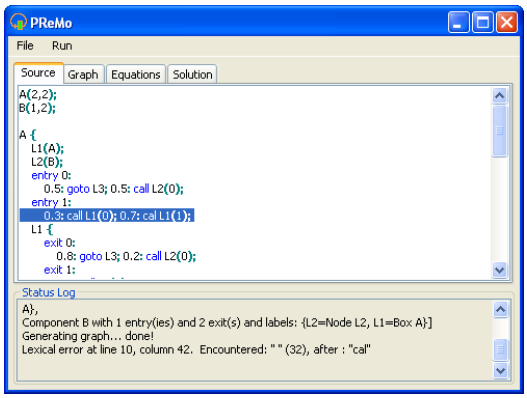

Figure 6: Highlighting the syntax error in the source code

After a successful parsing in the menu *Run*, new options will appear. We will be able to generate an equation system based on the model and in the case when our model is an RMC or an RSSG an option to generate the underlying transition graph will be enabled as well.

3.3 Graph generation and exporting as an image

If our model is an RMC or an RSSG then after successful parsing we will be able to generate the underlying transition graph as a dot input file from the

Run → *Generate graph* option. We can see an example on the fig. 7

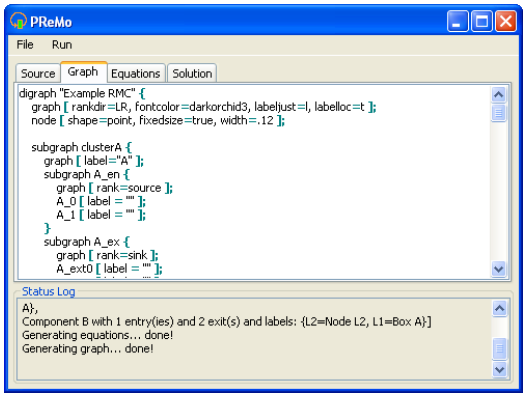

Figure 7: Generated graph source as a dot file

Once we generate a graph we can save it as a dot file for further processing (fig. 10) or if we have a dot processing system installed on our system eg. GraphViz, we can export as an image directly from the menu *File* → *Export as* → *PS* | *JPEG* | *GIF* | *PNG* int a desired image format.

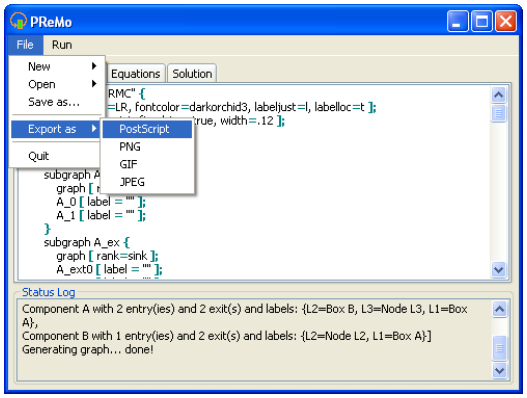

Figure 8: Exporting a graph as a image

3.4 Equation system generation

In the case of RMC and 1-exit RSSG the user by selecting *Run* → *Generate Equations* → *Termination probability* can generate a set of equations that the Least Fixed Point(LFP) solution gives exactly the probability of termination at any exit of all the components, starting at any node of the exit's component. Also for 1-exit RMC and 1-exit RSSG the user can generate an

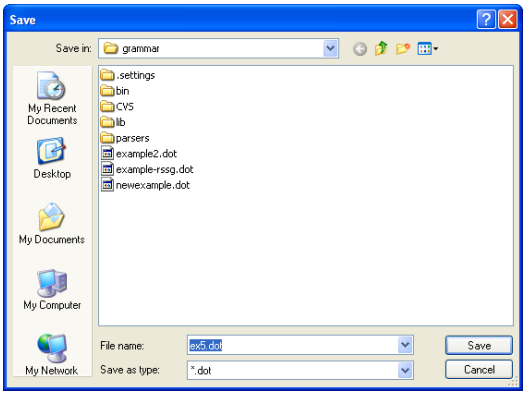

Figure 9: Saving a graph as a dot source file

equation system for which the LFP solution gives a solution to the problem of computing the average number of steps needed to terminate from a given node at the exit of the same component (and gives as an answer *Infinity* if the termination time is ∞).

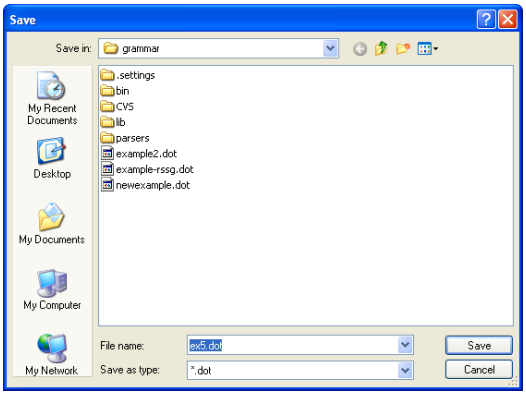

Figure 10: Saving a graph as a dot source file

3.5 Finding solution

We choose out of the submenu *Find solution* one of the available methods (fig. 11). After we have chosen one of them we will be asked for the tolerance (absolute, relative, or wont be asked if we are in a constant number of iteration mode, see subsection 3.8.2) (fig. 12). Depending on the equation system and the convergence criteria the solver can converge or not (fig. 13). If it converges, then it will list the value for each of the variables. Also, it will print some performance analysis data and the equation system statistics.

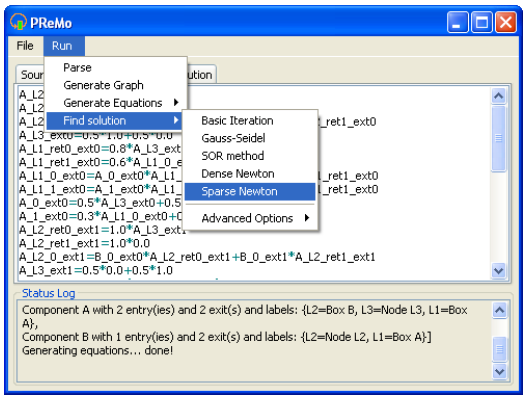

Figure 11: Choosing one of the methods of looking for a LFP solution

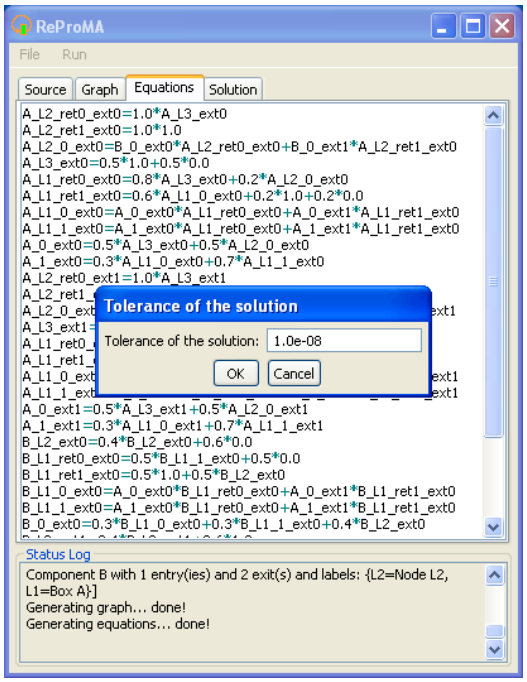

Figure 12: Specifying the tolerance of the solution

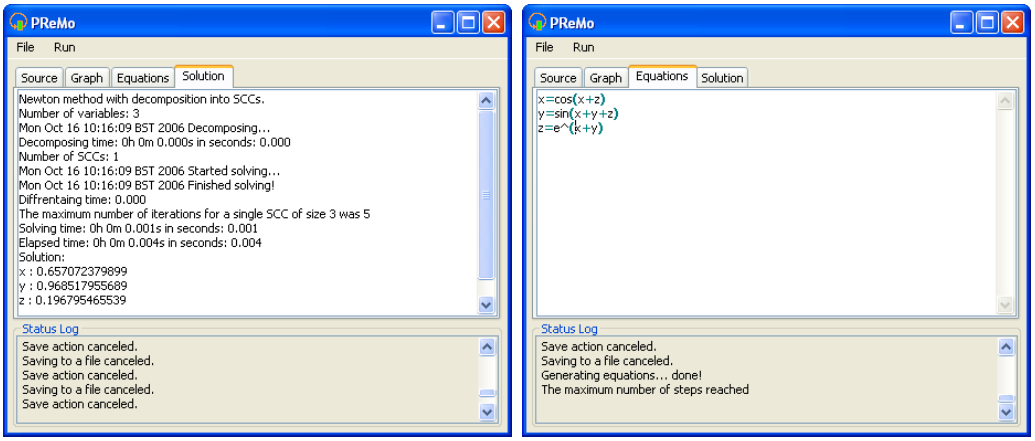

Figure 13: Solution was found on the left picture while on the right it wasn't as the maximum number of steps was reached

3.6 Interpreting the results

In case of an RMC and an RSSG the user has to interpret the variable names to get the solution to the problem. If we want to check the termination probability at exit j of the component `A`, starting at entry i of component `A` we have to look for the variable named: A_i_extj. In general the variable's names look like this: $componentName_$entryNum_ext$exitNum or for internal node `$label` of component `$componentName`: $componentName$label_ext$exitNum or for the exit numbered `$boxExitNum` of the box `$label` of the component `$componentName`: $componentName_$label_ret_$BoxExitNum_ext$exitNum. For example the solution to the RMC defined at the figure 1 looks like this:

```
Sparse Newton(IR) method with decomposition into SCCs.
Number of variables: 32
Tue Oct 31 15:07:51 GMT 2006 Decomposing...
Decomposing time: 0h 0m 0.021s in seconds: 0.021
Number of SCCs: 15
Tue Oct 31 15:07:51 GMT 2006 Started solving...
Tue Oct 31 15:07:51 GMT 2006 Finished solving!
Diffrentaing time: 0.004
The maximum number of iterations for a single SCC of size 18 was 5
Solving time: 0h 0m 0.333s in seconds: 0.333
Elapsed time: 0h 0m 0.392s in seconds: 0.392
Solution:
A_L2_ret0_ext0 : 0.500000000000
A_L2_ret1_ext0 : 0.500000000000
A_L2_0_ext0    : 0.500000009906
A_L3_ext0      : 0.500000000000
A_L1_ret0_ext0 : 0.500000002376
A_L1_ret1_ext0 : 0.00000000000
A_L1_0_ext0    : 0.250000003849
A_L1_1_ext0    : 0.0576923091901
A_0_ext0       : 0.500000005940
A_1_ext0       : 0.115384618103
A_L2_ret0_ext1 : 0.500000000000
A_L2_ret1_ext1 : 0.500000000000
```

```
A_L2_0_ext1    : 0.500000009906
A_L3_ext1      : 0.500000000000
A_L1_ret0_ext1 : 0.500000002376
A_L1_ret1_ext1 : 1.00000000000
A_L1_0_ext1    : 0.750000009348
A_L1_1_ext1    : 0.942307718159
A_0_ext1       : 0.500000005940
A_1_ext1       : 0.884615411760
B_L2_ext0      : 1.00000001100
B_L1_ret0_ext0 : 0.500000000000
B_L1_ret1_ext0 : 1.00000000550
B_L1_0_ext0    : 0.750000010997
B_L1_1_ext0    : 0.942307722769
B_0_ext0       : 0.907692327467
B_L2_ext1      : 0.00000000000
B_L1_ret0_ext1 : 0.500000000000
B_L1_ret1_ext1 : 0.00000000000
B_L1_0_ext1    : 0.250000002749
B_L1_1_ext1    : 0.0576923089363
B_0_ext1       : 0.0923076937823
```

3.7 Gray shading

When we modify the source code of an RMC, RSSG or SCFG, but generated before that a graph, equation system or computed a solution for the previous version of the model, then all the tabs retain their contents, but their background becomes gray(fig. 14) meaning that they are not up to date with the sourcecode. The user can still save the contents of them to a file, but it is forced to *Parse* the file again before doing anything else as all the other options from the *Run* menu becomes not available.

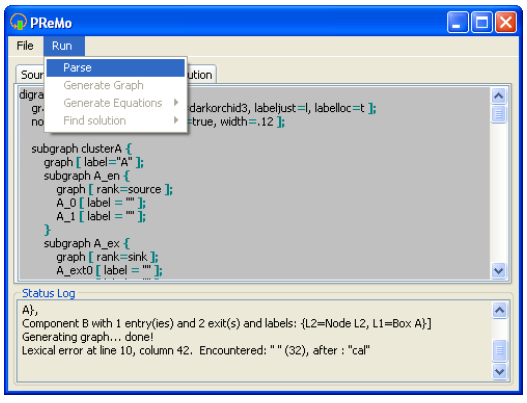

Figure 14: Gray shading the previously generated graph source after the sourcecode has been changed

3.8 Advanced options

All the options described here are accessible after selecting *Find solution* → *Advanced Options* from the *Run* menu.

3.8.1 Equations decomposition

The user can choose if he wants to decompose the equations or not in the case of using Jacobi, Gauss-Seidel or SOR methods as it is not necessary. It can choose whatever option he wants by checking or unchecking the mark next to the option *Decompose equations*. By default this option is checked.

3.8.2 Convergence Criteria

There are three different methods of declaring a convergence:

1. *absolute tolerance*, relative tolerance and running for a specified number of iterations. In the absolute tolerance mode algorithm stops when an absolute difference in any of the variables is lower than the threshold.

2. *relative tolerance* In this method I compute the difference between the new approximation and the old one for each of the variables and divide it by the previous(old) value of the variable (in case it is not equal to 0).

3. *number of iterations* Here we let the algorithm run for the specified number of steps no matter what absolute or relative error was obtained before that and return the approximation on the last step as the result of the computation.

3.8.3 Sparse linear solvers

The Sparse Newton's method uses at each step a sparse linear iterative solver to find the next approximation of the solutions. The user can choose from five solvers: BiConjugate Gradient (see `http://mathworld.wolfram.com/BiconjugateGradientMethod.html`), BiConjugate Gradient Stabilized (see `http://mathworld.wolfram.com/BiconjugateGradientStabilizedMethod.html`), Iterative Refinement (preconditioned Richardson method)(see `http://math.nist.gov/iml++/ir.h.txt` or [5]), Conjugate Gradients Squared (see `http://mathworld.wolfram.com/ConjugateGradientSquaredMethod.html`), Quasi Minimal Residual (see `http://mathworld.wolfram.com/Quasi-MinimalResidualMethod.html`).

According to the performance testing the Iterative Refinement(preconditioned Richardson method) is the fastest among them in most of the occurring cases. However if the system of equations hast the determinant of the Jacobian matrix close or equal to zero at the point were it has the LFP solution, this method is very slow. In such a case user should choose Quasi Minimal Residual method that doesn't have this problem and is just a little bit slower on the average.

4 Syntax of the input languages in the BNF format

Formal grammars of the input languages in the BNF are defined below. Characters in the (**bold** typeset are a part of the syntax of the input language.

4.1 Recursive Markov Chain grammar

$\langle RMC \rangle$ ::= $\langle ComponentDeclaration \rangle^+$ $\langle ComponentDefinition \rangle^+$
$\langle ComponentDeclaration \rangle$ ::= $\langle Id \rangle$ **(**$\langle Int \rangle$**,** $\langle Int \rangle$**);**
$\langle ComponentDefinition \rangle$ ::= $\langle Id \rangle$ **{** $\langle BoxDeclaration \rangle^*$ $\langle EntryNode \rangle^+$
($\langle NodeDefinition \rangle$ | $\langle BoxDefinition \rangle$)* **}**
$\langle EntryNode \rangle$ ::= **entry** $\langle Int \rangle$**:** $\langle Transition \rangle^+$
$\langle ExitNode \rangle$ ::= **exit** $\langle Int \rangle$**:** $\langle Transition \rangle^+$
$\langle Transition \rangle$ ::= $\langle Probability \rangle$**:** $\langle Jump \rangle$**;**
$\langle Jump \rangle$ ::= **goto** $\langle Label \rangle$
| **return** $\langle Int \rangle$
| **call** $\langle Label \rangle$**(**$\langle Int \rangle$**)**

$\langle BoxDeclaration \rangle$ ::= $\langle Label \rangle$**(**$\langle Id \rangle$**);**
$\langle NodeDefinition \rangle$ ::= $\langle Label \rangle$**(**$\langle Transition \rangle^+$**)**
$\langle BoxDefinition \rangle$ ::= $\langle Label \rangle$**{**$\langle ExitNode \rangle^+$**}**
$\langle Int \rangle$ ::= $(DIGIT)^+$
$\langle Probability \rangle$ ::= (**0** | **1**)**.**$(DIGIT)^+$
$\langle Id \rangle$ ::= $UPCASE\_LETTER(LETTER|DIGIT)^*$
$\langle Label \rangle$ ::= $UPCASE\_LETTER(LETTER|DIGIT)^*$

4.2 Recursive Markov Decision Processes and Recursive Simple Stochastic Gamees grammar

$\langle RMC \rangle$ ::= $\langle ComponentDeclaration \rangle^+$ $\langle ComponentDefinition \rangle^+$
$\langle ComponentDeclaration \rangle$ ::= $\langle Id \rangle$ (**(**$\langle Int \rangle$**,** $\langle Int \rangle$**);**
$\langle ComponentDefinition \rangle$ ::= $\langle Id \rangle$ **{** $\langle BoxDeclaration \rangle^*$ $\langle InputCase \rangle^+$
($\langle NodeDefinition \rangle$ | $\langle BoxDefinition \rangle$)* **}**
$\langle InputCase \rangle$ ::= **case** $\langle Int \rangle$**:** $\langle Transition \rangle^+$
$\langle OutputCase \rangle$ ::= **case** $\langle Int \rangle$**:** $\langle Transition \rangle^+$
$\langle Transition \rangle$ ::= $\langle Probability \rangle$**:** $\langle Jump \rangle$**;**
$\langle Jump \rangle$ ::= **goto** $\langle Label \rangle$
| **return** $\langle Int \rangle$
| **call** $\langle Label \rangle$**(**$\langle Int \rangle$**)**
$\langle BoxDeclaration \rangle$ ::= $\langle Label \rangle$**(**$\langle Id \rangle$**);**
$\langle NodeDefinition \rangle$ ::= $\langle Label \rangle$**(**$\langle Transition \rangle^+$**)**
$\langle BoxDefinition \rangle$ ::= $\langle Label \rangle$**(**$\langle Id \rangle$**)** **{**$\langle OutputCase \rangle^+$**}**
$\langle Int \rangle$ ::= $(DIGIT)^+$
$\langle Probability \rangle$ ::= (**0** | **1**)**.**$(DIGIT)^+$
$\langle Id \rangle$ ::= $UPCASE\_LETTER(LETTER|DIGIT)^*$
$\langle Label \rangle$ ::= $UPCASE\_LETTER(LETTER|DIGIT)^*$

4.3 Stochastic Context Free Grammar grammar

$\langle SCFG \rangle$ ::= $\langle Production \rangle^+$

$\langle Production \rangle ::= \langle NonTerminal \rangle - > \langle Probability \rangle \ (\langle NonTerminal \rangle | \langle Terminal \rangle)^*$
$\langle NonTerminal \rangle ::= LETTER(LETTER|DIGIT)^*$
$\langle Terminal \rangle ::= LETTER(LETTER|DIGIT)^*$
$\langle Probability \rangle ::= \mathbf{1.0}|\mathbf{0.}(DIGIT)^+$

References

[1] J. Esparza, A. Kucera, and R. Mayr. Quantitative analysis of probabilistic pushdown automata: Expectations and variances. In *LICS*, pages 117–126. IEEE Computer Society, 2005.

[2] K. Etessami and M. Yannakakis. Recursive markov chains, stochastic grammars, and monotone systems of nonlinear equations. In V. Diekert and B. Durand, editors, *STACS*, volume 3404 of *LNCS*, pages 340–352. Springer, 2005.

[3] K. Etessami and M. Yannakakis. Recursive markov decision processes and recursive stochastic games. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *ICALP*, volume 3580 of *LNCS*, pages 891–903. Springer, 2005.

[4] K. Etessami and M. Yannakakis. Efficient qualitative analysis of classes of recursive markov decision processes and simple stochastic games. In B. Durand and W. Thomas, editors, *STACS*, volume 3884 of *LNCS*, pages 634–645. Springer, 2006.

[5] R. S. Varga. *Matrix Iterative Analysis*. Prentice-Hall, Englewood Cliffs, New Jersey, 1962.