Data Structures and Algorithms 120

Lecture 2: File I/O

Curtin University of Technology

Department of Computing
Curtin University

Last updated: 7 March 2016

Copyright Warning

This Week

- Files and efficient I/O with files
- File streams in Java
- Tokenizers
- Reading and parsing CSV files
- Writing CSV files
- Text vs Binary
- Serialization

Why Files?

- RAM is volatile and private to an application, so it's not a good match for the following purposes:
  - Storing application data long-term (between runs)
  - Sharing information between applications
  - Reading in bulk data provided by a user
- In contrast, files stored on disk are (semi)permanent, can be shared between applications and can be manipulated by the user outside the application

File Input/Output

- Unlike RAM, files are effectively an input to and/or an output from the application
  - Hence the term File I/O, for Input/Output
- Three basic steps in file I/O: (applies to any platform)
  - Open the file
  - Read data from file and/or write data to file
  - Close the file

File I/O in Java

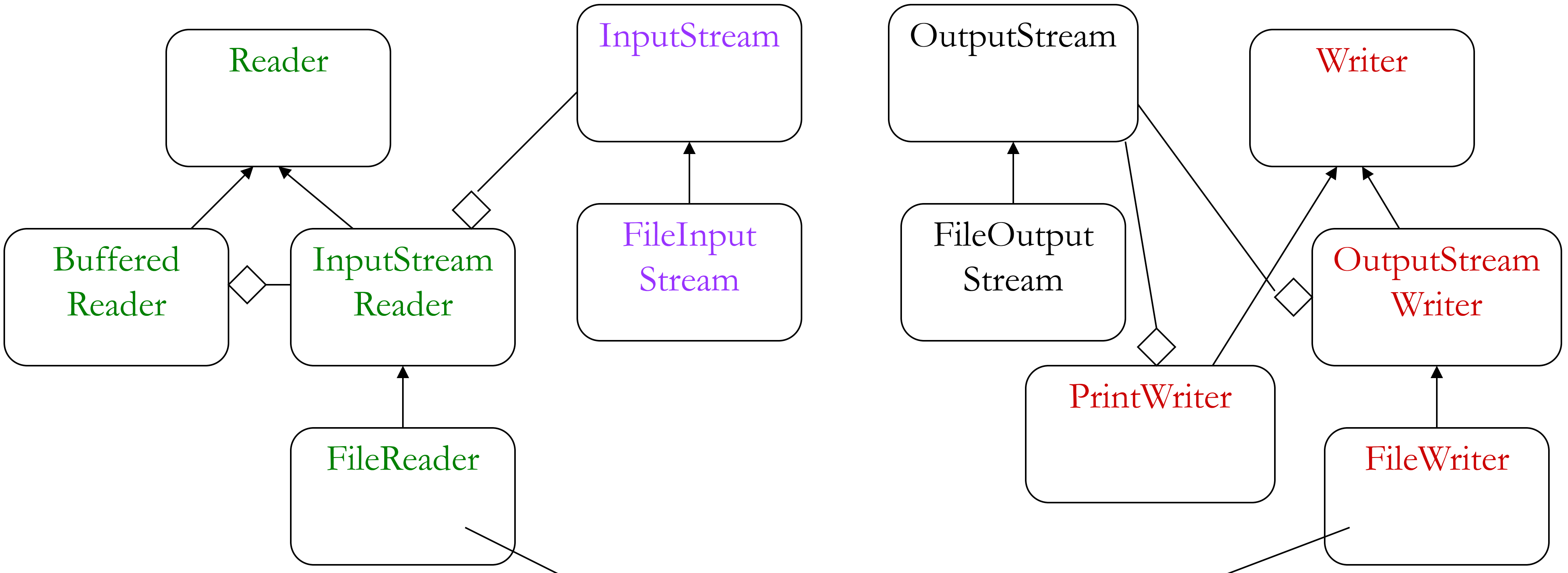
– Java encapsulates File I/O into a set of classes

- All file I/O classes support the three basic steps
- Different classes are used for manipulating different kinds of files or to perform I/O in different ways
  - » Depends on what you need
- Other kinds of I/O are also part of the same set of classes
  - » *e.g.*, keyboard input, screen output, network I/O, device I/O, *etc.*
  - » This was done to unify the handling of all the different I/O streams into a single consistent programming interface

Java I/O Classes

– The I/O classes can be split into three groups:
  - Classes that represent an I/O *stream* (*e.g.*, file, device, *etc.*)
  - Classes to read from a stream
  - Classes to write to a stream
– There are specialised versions of the basic classes for each I/O type (file, network, device, *etc.*)
– We will concentrate on the file I/O classes
  - I am just making you aware of the broader scope to help explain why there are so many classes involved!

Java I/O Class Hierarchy



These classes are more for convenience
and we won't be talking about them

Steps in Java for Reading a File

1. Create a stream object for a file
   - FileInputStream
2. Create an object that can read that stream
   - InputStreamReader
3. Read and process data from the file
4. Close the FileInputStream

– Note that a FileReader combines the first two steps into one

Steps in Java for Writing a File

Pretty much the same as for reading

1. Create a stream object for a file
   - FileOutputStream
2. Create an object that can write to that stream
   - OutputStreamWriter or PrintWriter
3. Write data to the file
4. Close the FileOutputStream

– Again, FileWriter combines the first two steps

Efficient File I/O

- If you look at the methods for InputStreamReader, you'll see that it has *two* `read()` methods
  - One for reading a byte at a time
    - This returns –1 if no more bytes to read (*i.e.*, end-of-file reached)
  - A second for reading an array of bytes at a time
- Since it's rare that we ever want just a single byte of data, the first method seems less useful
- However, when trying to read in text data there are conceptual difficulties with reading chunks at a time

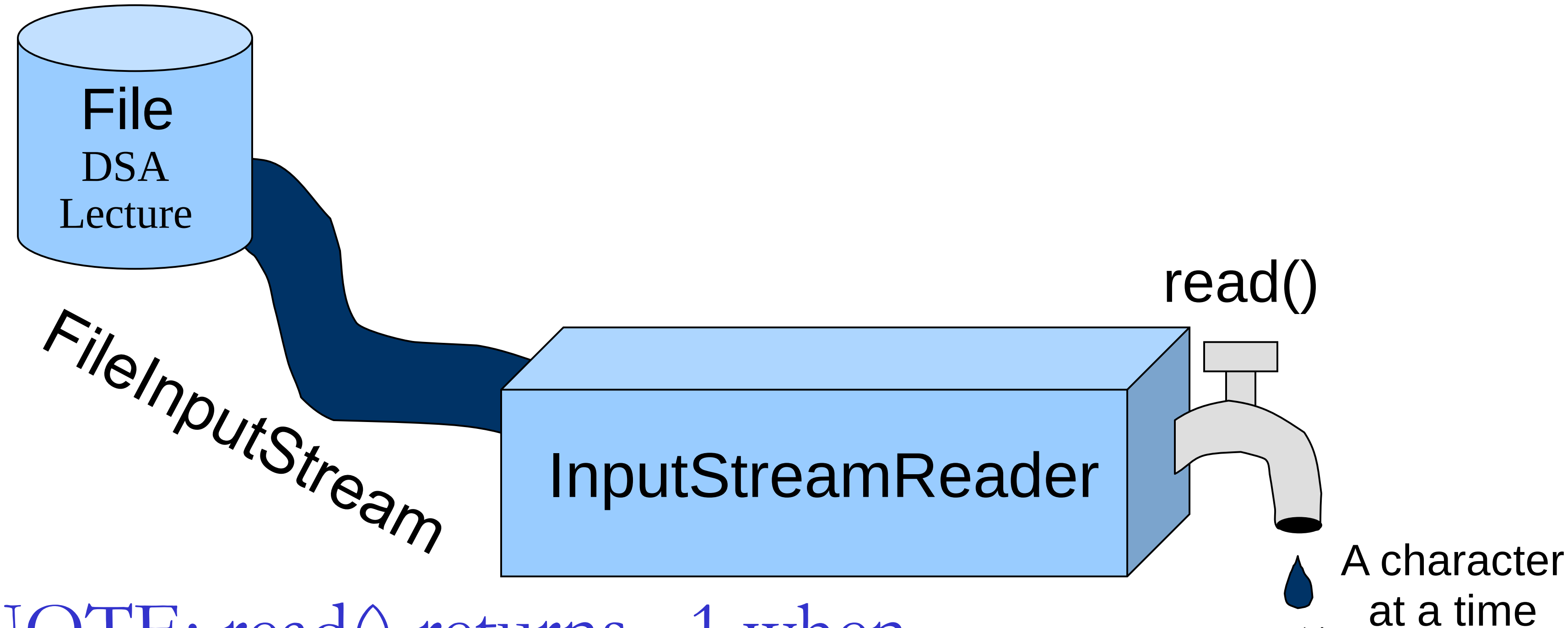
Text vs Binary Data

- Data files generally fall into two broad categories:
  - Files containing text data (relatively unstructured)
  - Files containing binary data (everything else)
- Binary data is highly structured
  - *e.g.*, images, databases, executable files, *etc.*
  - There is a lot of prior knowledge on precisely where information exists in the file and how large blocks are
- Text data is *un*structured, so it is usually impossible to know beforehand how many bytes to read
  - *e.g.*, how many characters in an arbitrary line of text?

Efficient File I/O

- So reading blocks of N bytes from a binary file is OK
  - You will know beforehand how many bytes are needed
- But when reading text, you have to constantly be ready for the end of data since you can't predict it
  - *e.g.*, end of word, end of line, or end of file
- One approach is to read data in a byte at a time, and check each byte for end-of-X (' ', '\n' or -1)
  - The problem is that this will be very slow since hard disks are fastest at reading blocks of data at a time
    - » It's like filling a bucket one drop at a time vs opening the tap and letting the water flow freely into the bucket

InputStreamReader



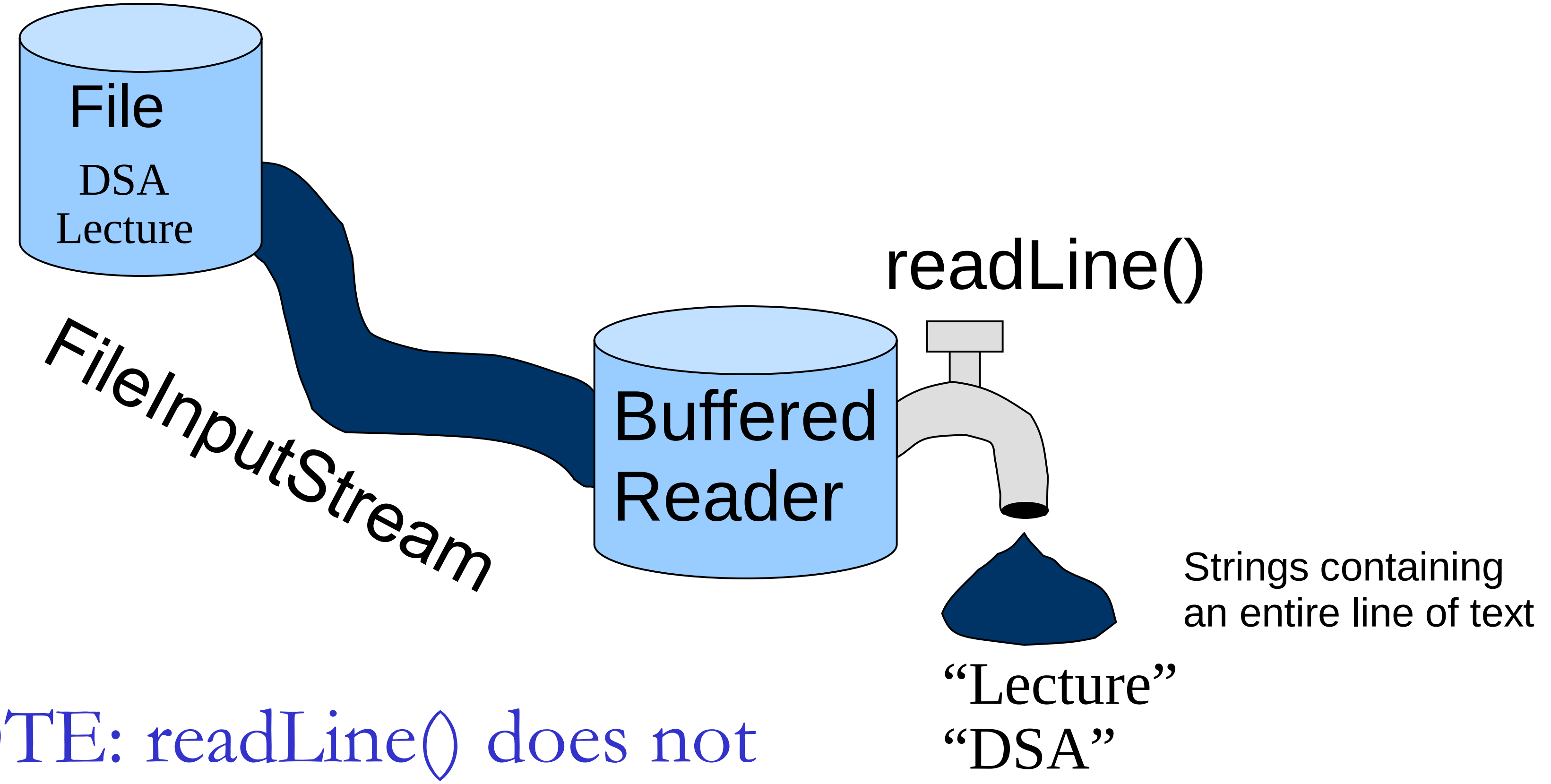
NOTE: read() returns –1 when end-of-file (ie: no more data) occurs

- Also note the ‘\n’ (newline) is returned; just like any other character in the file

BufferedReader

- Reading a file a single byte at a time is very slow.
- BufferedReader reads in chunks of data, *buffering* it in memory to search for end-of-X
  - » In particular, the readLine() method looks for end-of-line
  - The idea is to read in ('buffer') a chunk of (say) 1024 bytes, then search these bytes in RAM one at a time for the '\n'
    - » \n = newline character, marking the end of a line of text
    - » Since RAM is much faster than disk, this is much more efficient
  - When a line is found it is extracted and returned, but the rest of the buffer is *kept in memory*
    - » Then for subsequent readLine() calls, it first checks the buffer for the '\n' before reading another 1024-byte chunk from the file
  - Note: you create a BufferedReader from another Reader

BufferedReader



NOTE: readLine() does not include the ‘\n’ in the returned lines and returns null when end-of-file occurs

File Reading: One Line at a Time

```
METHOD readFileExample IMPORT filename EXPORT NOTHING

theFile ← OPENFILE filename
lineNum ← 0
INPUT line FROM theFile

WHILE NOT (theFile = EOF)          ← EOF = end of file. Detecting this is very language-specific
   lineNum ← lineNum + 1
   processLine(line)               ← Whatever processing on the line is required

   INPUT line FROM theFile         ← Read the next line
ENDWHILE

CLOSEFILE theFile                  ← Close the file

ENDMETHOD
```

File Reading: One Line at a Time

```
private void readFileExample(String inFilename) {
   FileInputStream fileStrm = null;
   InputStreamReader rdr;
   BufferedReader bufRdr;
   int lineNum;
   String line;
   try {
      fileStrm = new FileInputStream(inFilename);        ← Open the file
      rdr = new InputStreamReader(fileStrm);             ← Create a reader to read the stream
      bufRdr = new BufferedReader(rdr);                  ← To read the stream one line at a time

      lineNum = 0;
      line = bufRdr.readLine();                          ← Read the first line
      while (line != null) {                             ← While not end-of-file, process and read lines
         lineNum++;
         processLine(line);                              ← Whatever processing on the line is required
         line = bufRdr.readLine();                       ← Read the next line
      }
      fileStrm.close();                                  ← Clean up the stream
   }
   catch (IOException e) {                               ← MUST catch IOExceptions
      if (fileStrm != null) {                            ← Clean up the stream if it was opened
         try { fileStrm.close(); } catch (IOException ex2) { } // We can't do anything more!
      }
      System.out.println("Error in file processing: " + e.getMessage());   ← Or do a throw
   }
}
```

Notes on File Handling

– Make sure you close the file as soon as possible

- The operating system must track what files are open
  - » The O/S remembers where you were in the file, etc
- The resources available for this tracking are limited
  - » Run out and the O/S will terminate your program
- Thus don't leave files open – clean them up *early*
  - » Java doesn't free objects immediately – it waits for the garbage collector, so *always* explicitly close() a file once finished with it

Notes on File Handling

- You *must* handle IOException
  - IOExceptions must be caught
    - Checked Exception
    - The compiler will complain otherwise
    - This also forces us to do the try..catch around close()
  - Note that you can add a `throw IOException` clause to the method to 'avoid' having to catch IOExceptions
    - It just means that now the *calling* method must catch them

Parsing

- When dealing with text, it's often necessary to take it apart and organize it ready for processing/storage
- This is called parsing – to determine and extract the structure of a piece of text
  - The word originally comes from syntax analysis of written language
- Examples of where parsing is needed:
  - Natural language processing (*e.g.*, spelling/grammar checks)
  - Building Web search indexes
  - Compilers: must parse code to detect stmts and variables

Tokenizing

- Tokenizing is the first step in parsing: the process of breaking up a stream of text into basic elements
  - » We'll use US spelling (with a 'z') to avoid confusion
  - These elements are called tokens, and what they are depends on what the application is parsing
    - » *e.g.*, Single words, entire lines, equation terms, *etc.*
  - Tokens are broken up by searching for character(s) that *delimit* the boundary of a token
    - » *e.g.*, lines are separated by a '\n' newline character
    - » *e.g.*, words are separated by spaces, commas and periods
    - » *e.g.*, equation operands are separated by operators +/*-

Tokenizing with Java

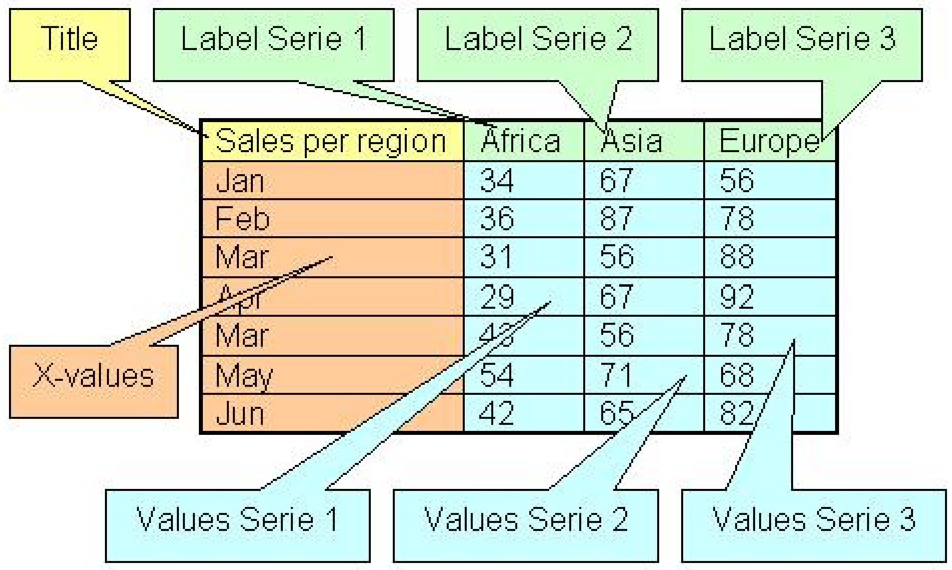
- Java provides two classes to assist in tokenizing
  - StringTokenizer
  - StreamTokenizer
  - They do a similar job except one works on Strings while the other works on Readers
    - » We'll focus on StringTokenizer
- Java also provides the split(String regex) method in the String class for simple tokenizing of a String
  - It's easier to use than the tokenizers, but won't return the delimiting characters (which can be needed)
  - and you don't know what a regex is!

Comma Separated Values

- We'll take a little detour and introduce comma separated values
  - We'll use this as an example for StringTokenizer later on
- We often need to store data to a file
  - The question is, in what form should we store it?
- If the data is in table or matrix form, one can write it out as a set of rows and columns in a certain format:
  - One row is written per line
  - Each row contains multiple fields, one per column
  - Each field's value is separated by a comma ','

CSV Example

Content of File




```
Sales per region,Africa,Asia,Europe
Jan,34,67,56
Feb,36,87,78
Mar,31,56,88
Apr,29,67,92
Mar,43,56,78
May,54,71,68
Jun,42,65,82
```

The CSV File

CSV Notes

- The delimiting commas have no trailing space
  - It might look better to have a space after the comma, but it just makes it harder to parse when reading the file later!
- Column sizes don't have to be consistent across rows
  - *e.g.*, the first row (headings) has much longer fields than the same columns in subsequent rows
    - » It would be a waste of space to pad out the fields
    - » Parsing/tokenizing must handle these variable-length fields
- Numeric data is converted into its textual equivalent
  - If we saved integers directly, we might get things that look like text '\n' (ASCII 13) or ',' (ASCII 44) but are merely part of the data

Text vs Binary

- You don't *have* to save table data in CSV format
  - In fact, dumping raw binary data is often more efficient, and you know how large each field is (*e.g.*, ints = 4 bytes)
- However, CSV text data has some advantages:
  - Easy for humans to read and edit
  - Highly portable to different platforms
    - » big endian vs little endian
  - Fields are explicitly separated with commas
  - CSV is a widely-known format
    - » XML is now beginning to supplant CSV as the standard format for data interchange between companies, but CSV is still used

StringTokenizer and CSV Data

- So let's parse a CSV file with a StringTokenizer
  - StringTokenizer works a lot like an iterator (unfortunately, it's from JDK 1.0 and so isn't an actual Iterator – it just has similar usage)
  - The idea is to create a StringTokenizer based on the string to parse
  - Then repeatedly ask the StringTokenizer for the next token, until no more tokens exist
    - » And for each token, perform appropriate processing
  - Note that we need to create a new StringTokenizer for every line that is read from the CSV file
    - » So we'll put the tokenizing code into the processLine() method

Parsing a Single CSV Row

```
private void processLine(String csvRow) {
   String thisToken = null;
   StringTokenizer strTok;

   strTok = new StringTokenizer(csvRow, ",");        ← Initialise tokenizer to parse the csvRow
   while (strTok.hasMoreTokens()) {                   ← Iterate over tokens
      thisToken = strTok.nextToken();
      System.out.print(thisToken + " ");              ← Print out each column for this row
   }
   System.out.println("");
}
```

Output

– Given the following CSV data file:

```
97452,James,88,96,82,86
99576,Alan,6,46,34,38
99888,Geoff,100,68,72,75
```

– The following would be the output of having readFileExample() call the shown processLine():

```
97452 James 88 96 82 86
99576 Alan 6 46 34 38
99888 Geoff 100 68 72 75
```

Writing Text Files

- Writing files is actually simpler than reading them
  - Since you don't have to worry about parsing!
- The overall approach is the same:
  - Open a FileOutputStream
  - Create a Writer
  - Output data to the file using the Writer
- One thing to be careful with is that you must ensure that newlines and commas are put in the right place
  - Assuming you are outputting CSV format, of course!

PrintWriter

- You can use the basic OutputStreamWriter, but it's a bit clunky
  - The write() method requires an array of bytes, so you have to copy your data into an array before writing
- It would be easier if you could write to a file in the same way that you can print messages to the console
  - Fortunately you can: PrintWriter
  - In fact, `System.out` *is* an instance of a PrintWriter!
  - Thus writing to file can be identical to printing to screen
    - Just don't forget to close() the file after you are done!!

PrintWriter Example

- The next slide shows an example of writing to a file
- It assumes you have passed in the student marks info that was read in from the earlier CSV file
  - *e.g.*, we are saving one line of the data to a new CSV file
  - Note that the commas are inserted between the fields, and there are no spaces added

Writing a CSV Row

```
private void writeOneRow(String filename, int ID, String name, double assign, double test,
                              double exam, double overall) {
   FileOutputStream fileStrm = null;
   PrintWriter pw;

   try {
      fileStrm = new FileOutputStream(filename);        ← Open the file for writing
      pw = new PrintWriter(fileStrm);                   ← Initialise writer

      pw.println(id + "," + name + "," + assign + "," + test + "," + exam + "," + overall);
      pw.close();                                       ← Clean up the stream
   }
   catch (IOException e) {                    ← MUST catch IOExceptions
      if (fileStrm != null) {                 ← Clean up the stream if it was opened
         try { fileStrm.close(); } catch (IOException ex2) { } // We can't do anything more!
      }
      System.out.println("Error in writing to file: " + e.getMessage());    ← Or do a throw
   }
}
```

Text vs Binary

- Text advantages over binary:
  - Human-readable and editable
  - Text representation of numbers, strings are unambiguous
  - Slight variations in format can often be handled
    - » *e.g.*, if a space is added after the ', ' in CSV it could be ignored
- Binary advantages over text:
  - Smaller size for storing the same data
    - » *e.g.*, pi = "3.141592653589793" in text (17 bytes), but pi can be stored with similar precision in only 8 bytes using a double
  - Faster – no need to convert numbers to/from text strings
  - Simpler to parse since exact format and sizing of each piece of data is known in advance

ContainerClass

```
public class ContainerClass
{
   private String name;
   private int serial;
   private char type;
   private double weight;
   private Location location;

   //normal constructors, getters and setters
   //not relevant to this example

   void open(String fileName);
   void save(String fileName);
}
```

Writing Binary Data

– DataOutputStream
  - Created from a FileOutputStream
  - Includes the following methods
    - » void writeUTF(String s);
    - » void writeInt(int ii);
    - » void writeChar(char ch)
    - » void writeDouble(double d);
    - » void flush();
    - » void close();
    - » and many more

ContainerClass save method

```
private void save(String filename){
   FileOutputStream fileStrm;
   DataOutputStream dataStrm;

   try {
      fileStrm = new FileOutputStream(filename);  ← Underlying stream
      dataStrm = new DataOutputStream(fileStrm);  ← Data stream

       dataStrm.writeUTF(name);
      dataStrm.writeInt(serial);
      dataStrm.writeChar(type);
      dataStrm.writeDouble(weight);                ← save binary data to filename


      fileStrm.close();                            ← Clean up
   }

   catch (IOException e){
      if (fileStrm != null) {                      ← Clean up the stream if it was opened
         try { fileStrm.close();
         }
         catch (IOException ex2) { }               ← We can't do anything more!
      }
      System.out.println("Error in file processing: " + e.getMessage());    ← Or do a throw
   }
}
```

Reading Binary Data

- DataInputStream
  - Created from FileInputStream
  - Includes the following methods:
    - » char readChar();
    - » double readDouble();
    - » int readInt();
    - » string readUTF();
    - » void close();
    - » and many more

ContainerClass open(String fileName)

```
private void open(String filename){
   FileInputStream fileStrm;
   DataInputStream dataStrm;


   try {
      fileStrm = new FileInputStream(filename);          ←  Underlying stream
      dataStrm = new DataInputStream(fileStrm);          ←  Data stream

      this.name = dataStrm.readUTF();                    ←  read data into classfields
      this.serial = dataStrm.readInt();
      this.type = dataStrm.readChar();
      this.weight = dataStrm.readDouble();

      fileStrm.close();                                 ← Clean up
   }
   catch (IOException e){
      if (fileStrm != null) {                           ← Clean up the stream if it was opened
         try { fileStrm.close();
         }
         catch (IOException ex2) { }                    ← We can’t do anything more!
      }
      System.out.println("Error in file processing: " + e.getMessage());    ← Or do a throw
   }
}
```

Serialization

- Persistence
- Text vs binary for serialization
- Java's object serialization mechanism
- Serialization streams

Persistence

- Persistence: The saving and loading application data so that it lasts between executions of the application
  - *i.e.*, the task of storing an application's state to disk and later loading it back to continue running
- We've already had a look at using text and binary formats to save data to disk and read it in later
  - Loading may involve parsing (text) saved data to translate it into a form that is suitable for manipulation by the application
  - Not all persistence requires parsing, but most do simply because storage is static whereas manipulation is dynamic

The Need for Serialization

- What if ContainerClass has a classfield which is an object of another class (aggregation)?
  - it does! Location class
- What if ContainerClass had an inheritance relationship that has classfields in the super class?

Serialization Issues

– Every Java object is a reference (pointer) – even Strings!
  - Pointers let you jump around in memory (*i.e.*, non-contiguous)
    - » This is so useful that we simply cannot do without pointers!
  - But you cannot save a pointer and expect it to be useful after re-loading it since the pointer says where the data *used to be* in the *old* run of the application!
    - » When the application runs again, it won't place objects in the same place – every time new is called, it returns a different address
    - » Furthermore, the object doesn't even exist yet – it is about to be loaded from disk!
  - In other words, pointers cannot be made persistent

How Does Serialization Work

- So how could it possibly work?
  - The idea is pack the data to be saved into a single contiguous block so that it can be written all together
  - Java is able to do this *automatically* via a language feature called reflection
- Reflection is the ability for an object to query the methods and fields of another class *at runtime*
  - This returns the names of fields, methods, parameters, return types, etc, and the ability to actually set/get the value of each field
    - » Thus Java can figure out what data fields an object has and pack them into a byte array for saving

Java Serialization

- writing your own serialization code in Java is quite involved
  - The serialization buffer must be an array of bytes
  - Then you must copy every byte of an object into the serialization byte array
- This is why Java's serialization mechanism is so very useful
  - It does all the work of formatting and copying for you
  - From the programmer's viewpoint, it all comes down to the Serializable interface and a couple of Stream classes

Serializable Interface

- To let Java serialize your class, simply make it inherit from the Serializable interface *e.g.*,

```
public class ContainerClass implements Serializable
{
   ...
}
```

  - That's it! **You do not have to write any code** – Java will determine the contents of your class and serialize it for you *entirely automatically* using reflection
  - It is possible to override Java's serialization if needed
- Note that Serializable is an *empty* interface! In Java:

```
public interface Serializable { }
```

  - *i.e.*, it is only used to mark a class as serializable for Java

Object I/O Streams

- Any class that implements Serializable can then be written and read via two serialization-specific classes
  - ObjectOutputStream – for serializing an object and writing it out to an underlying stream
  - ObjectInputStream – for reading an underlying stream, deserializing the object and returning it
    - » This creates the object and sets its fields to the serialized data
  - The two classes both know the serialization format
- When creating an Object stream, you must provide another stream – the underlying 'physical' stream
  - Usually FileOutputStream and FileInputStream

Object I/O Streams

- ObjectOutputStream will serialize the given object *and all objects referenced by that object*
  - This is done recursively, so that the entire object tree will be serialized
  - This supports the composition relationship: a car object consists of wheel objects, engine object and chassis object
    - » So when serializing a car, we must serialize all contained objects
  - Objects not marked as Serializable won't be serialized
    - » You can also add in custom serialization/deserialization steps, but this is only needed in special circumstances
- ObjectInputStream will deserialize all contained objs

Serializing an Object

```
private void save(ContainerClass objToSave, String filename)
{
   FileOutputStream fileStrm;
   ObjectOutputStream objStrm;

   try {
      fileStrm = new FileOutputStream(filename);       ← Underlying stream
      objStrm = new ObjectOutputStream(fileStrm);      ← Object serialization stream
      objStrm.writeObject(objToSave);                  ← Serialize and save to filename
                                                         This will also save the ContainerClass'
                                                         contained Location object
      objStrm.close();                                 ← Clean up
   }
   catch (Exception e) {                              ← should do more here
      throw new IllegalArgumentException("Unable to save object to file");
   }
}
```

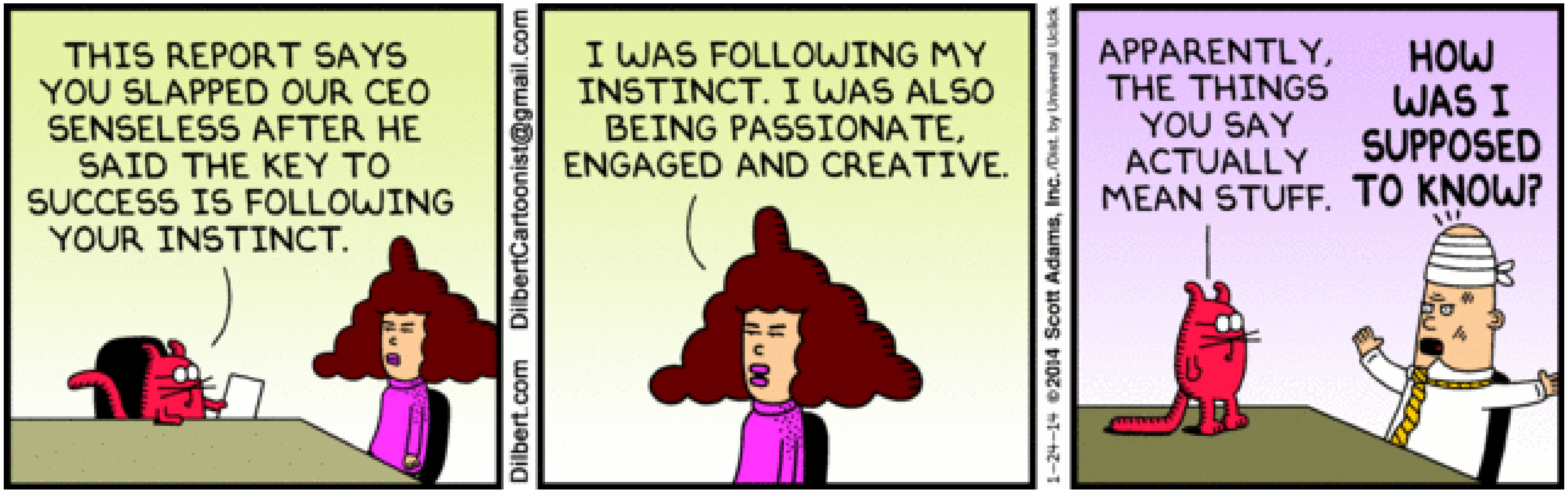
Deserializing an Object

```
private ContainerClass load(String filename) throws IllegalArgumentException
{
   FileInputStream fileStrm;
   ObjectInputStream objStrm;
   ContainerClass inObj;

   try {
      fileStrm = new FileInputStream(filename);        ← Underlying stream
      objStrm = new ObjectInputStream(fileStrm);       ← Object serialization stream
      inObj = (ContainerClass)objStrm.readObject();    ← Deserialize. Note the cast is needed
      objStrm.close();                                 ← Clean up

   }
   catch (ClassNotFoundException e) {
      System.out.println(“Class ContainerClass not found” + e.getmessage);
   }
   catch (Exception e) {
      throw new IllegalArgumentException("Unable to load object from file");
   }
   return inObj;
}
```

XML Serialization

- The Object<Input|Output>Stream classes use a format that is specific to Java
  - Don't expect to be able to pass the object to another language and deserialize it, even if the classes are identical!
    - In fact, backwards compatibility *in Java* has even been broken
- An alternative that is more portable is XML format
  - XML has become a defacto standard for serializing
  - Java provides XMLEncoder and XMLDecoder for this, which work similarly to the Object<Input|Output>Stream classes
  - JAXB (Java Architecture for XML Binding) is a more sophisticated alternative for XML, but also more complex

Not The End



Stay tuned for an overview of network reliability.

OBDD-H

SEP Team 7 (2015)

Dani Brad Josh & Paul

The Aim

- Create a network reliability calculator with the OBDD-H algorithm
- Calculations will be done on graph files as text representations of networks
- Provide a convenient interface to interact with the algorithm

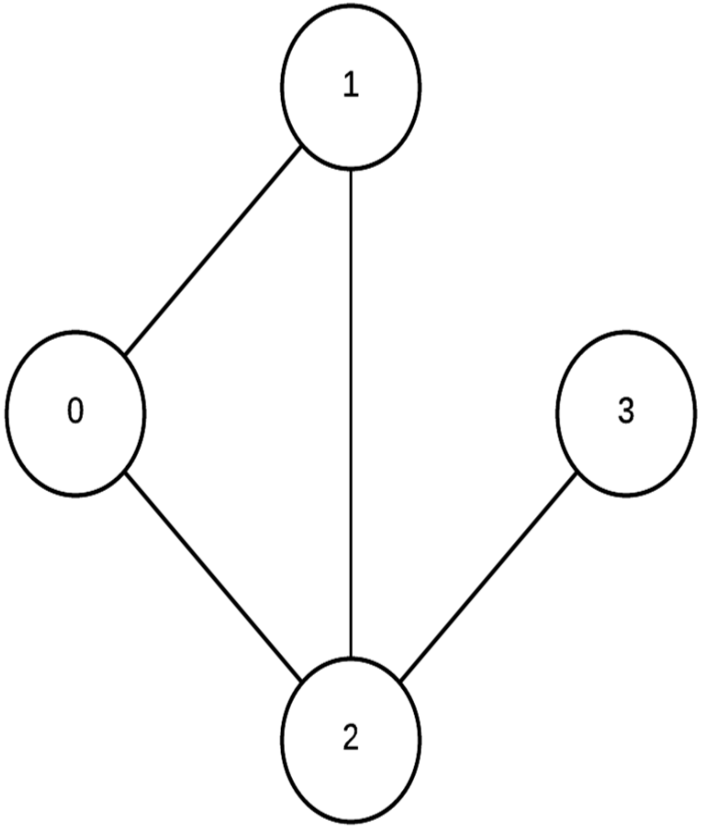
Reliability Calculation

- Ordered Binary Decision Diagram - Hybrid
- Vertices of a network are considered valid
- Edges have a given reliability
- The network has a given source and sink

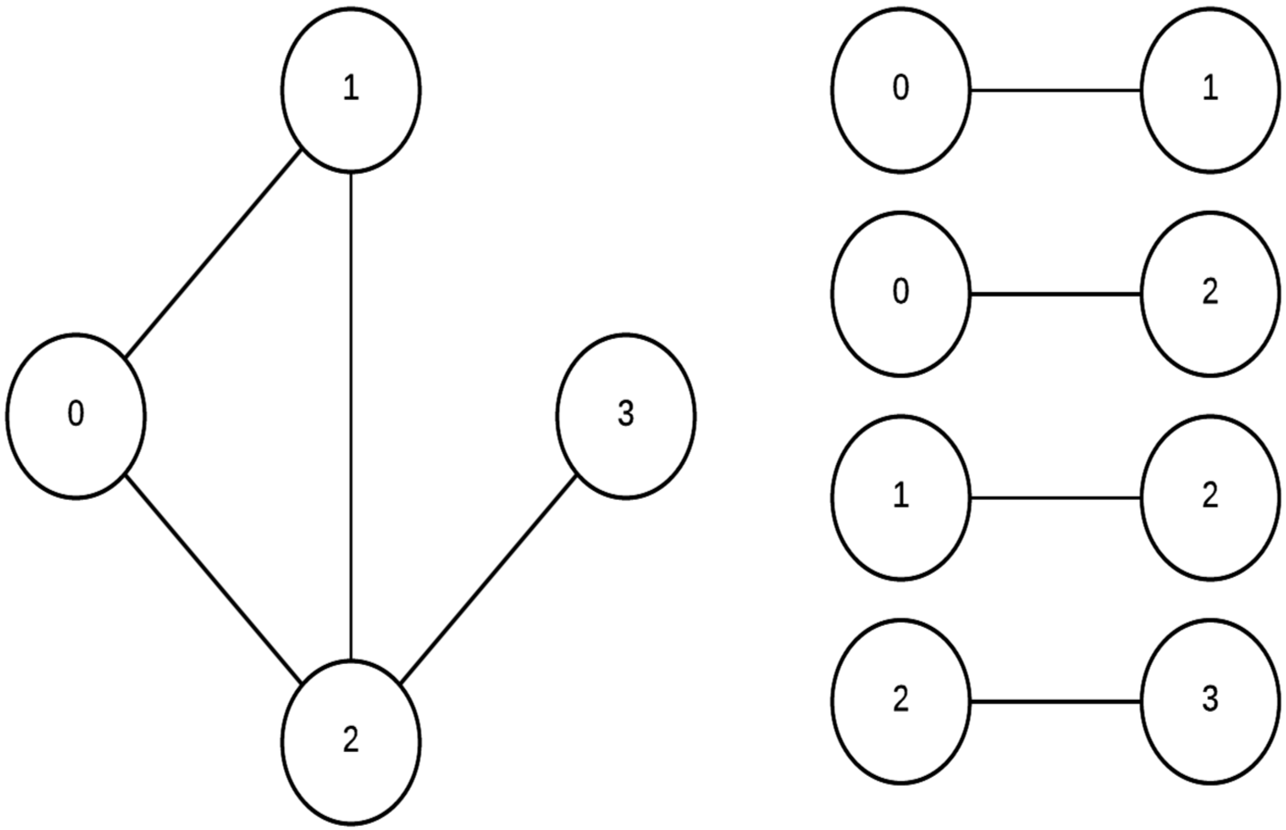
Reliability Calculation

- Construct a binary tree edge by edge
- Each leaf represents a permutation of the state of the problem
- Add the reliabilities of nodes with an active path from source to sink

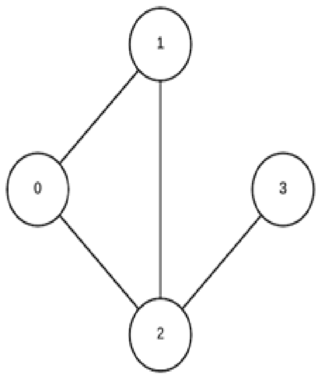
Example Network Graph

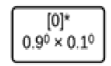


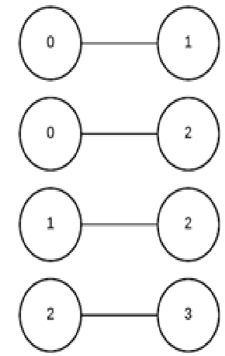
Edge Representation



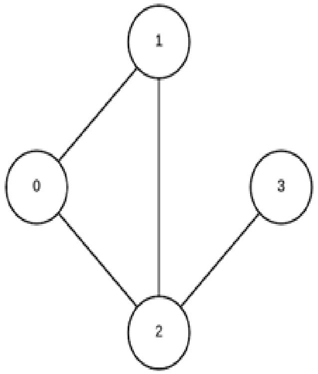
Source node created

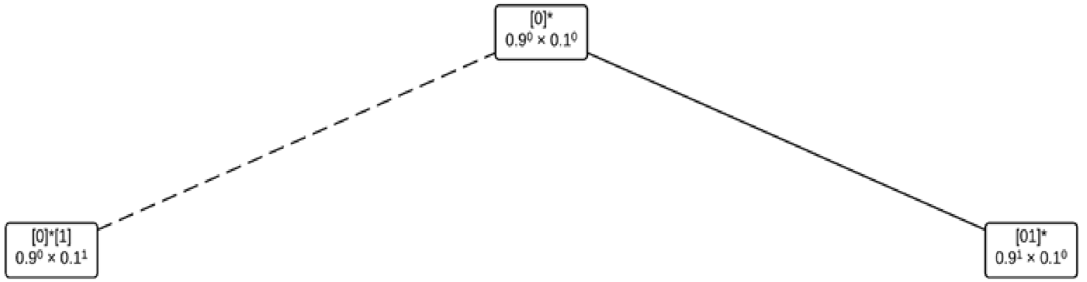


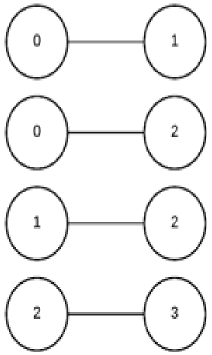




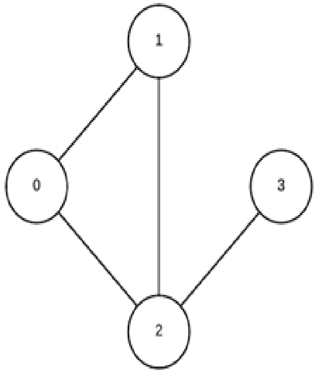
Nodes for edge 0-1 added

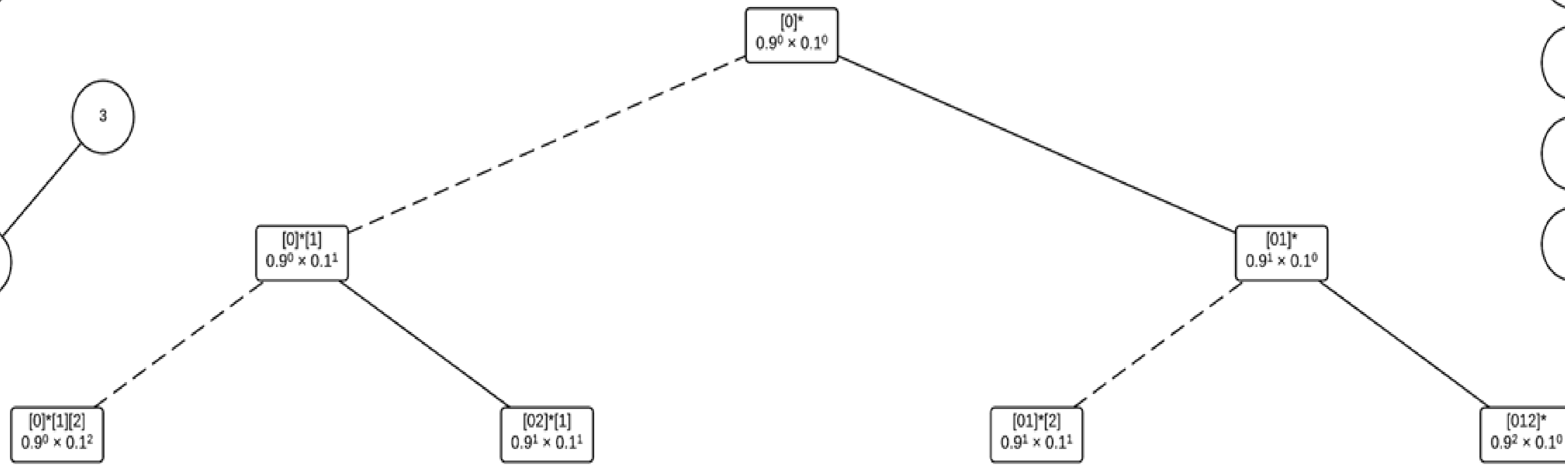


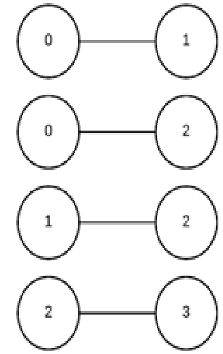




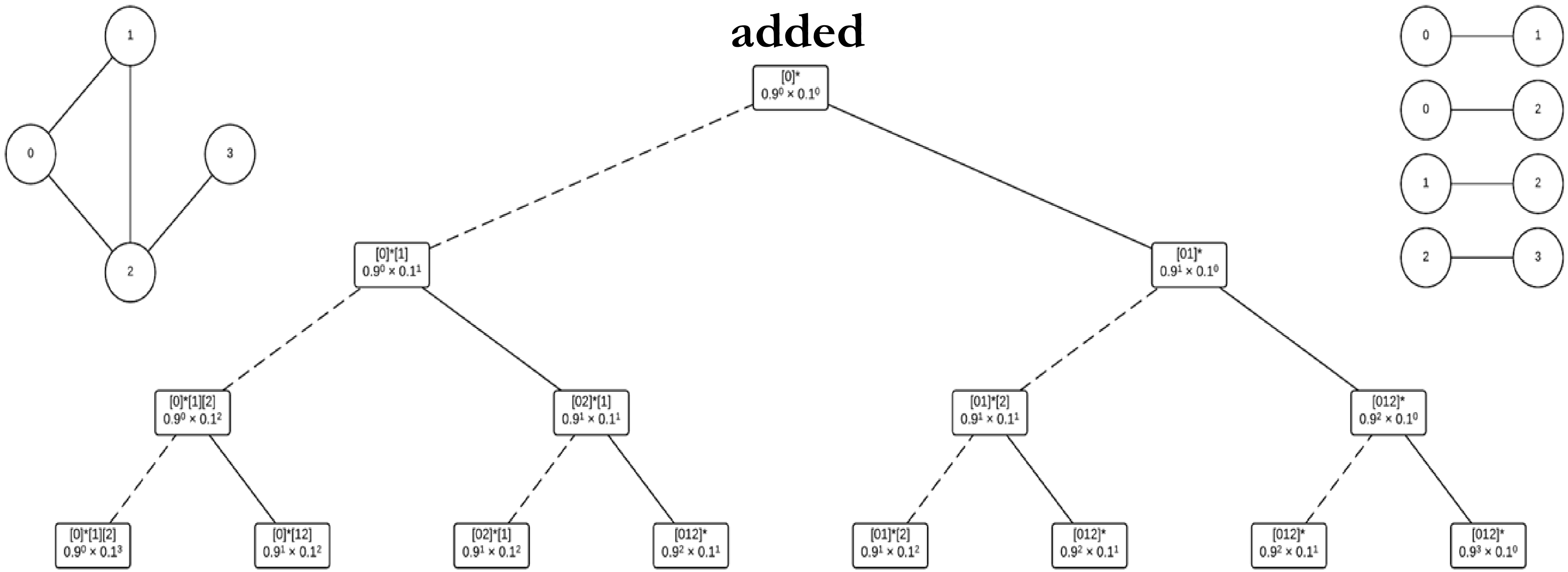
Nodes for edge 0-2 added



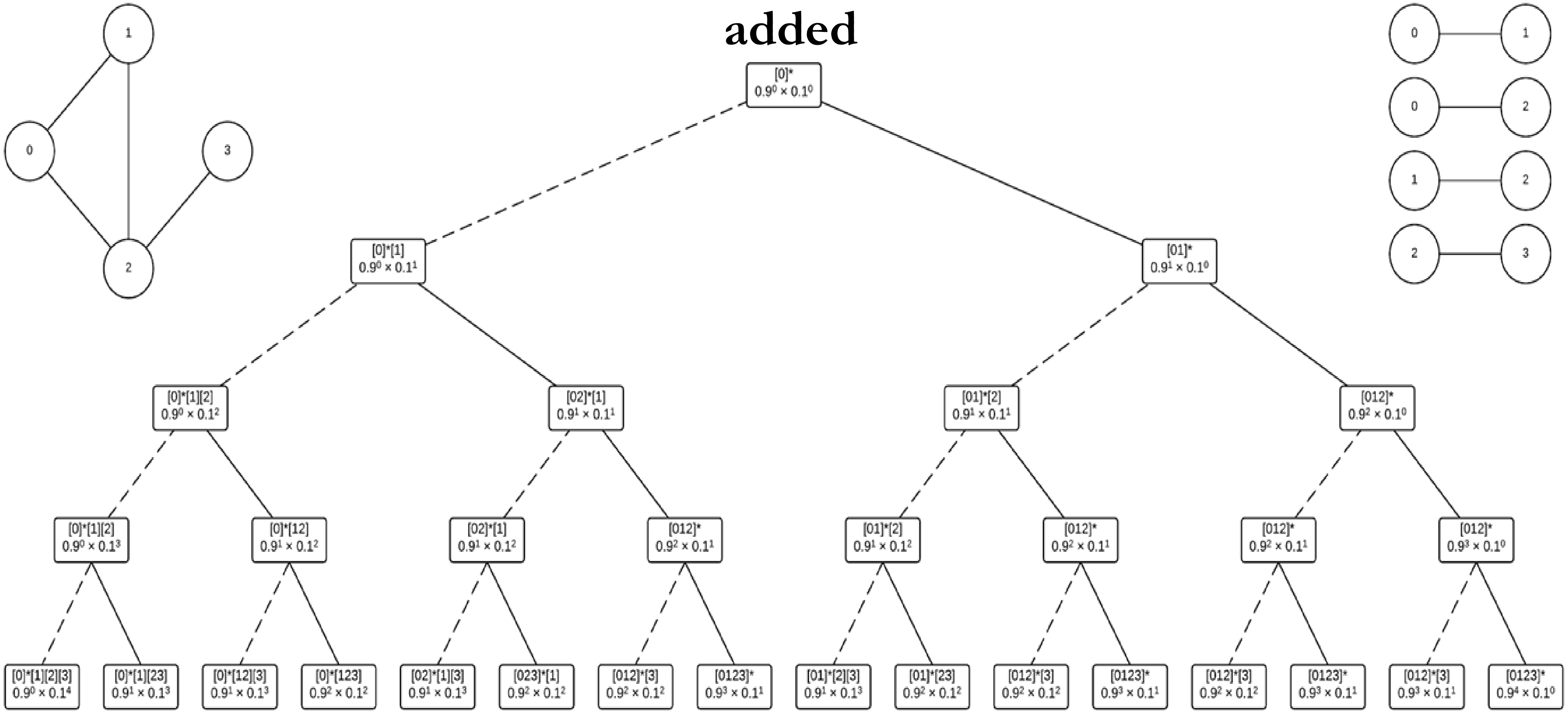




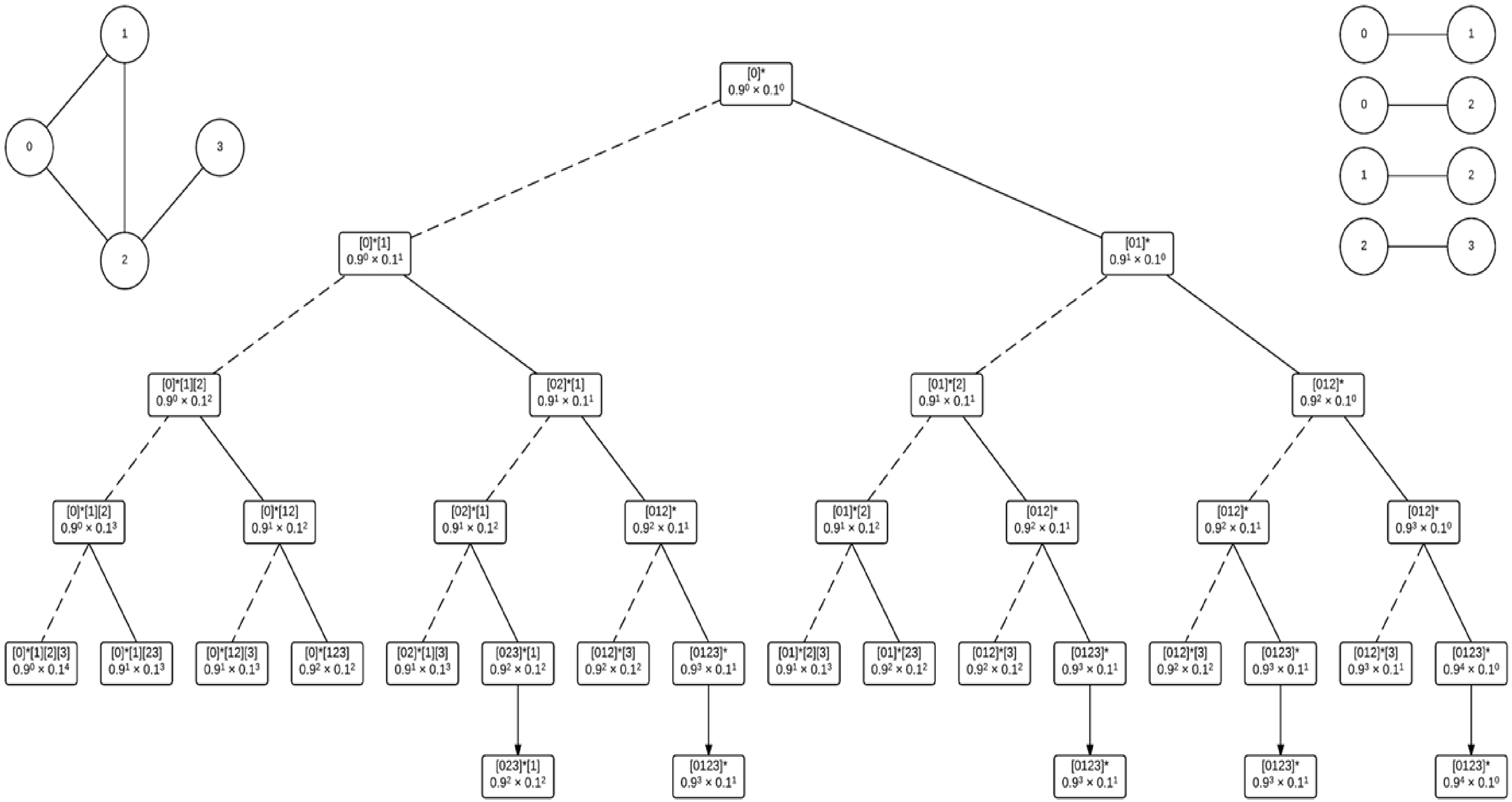
Nodes for edge 1-2 added



Nodes for edge 2-3 added

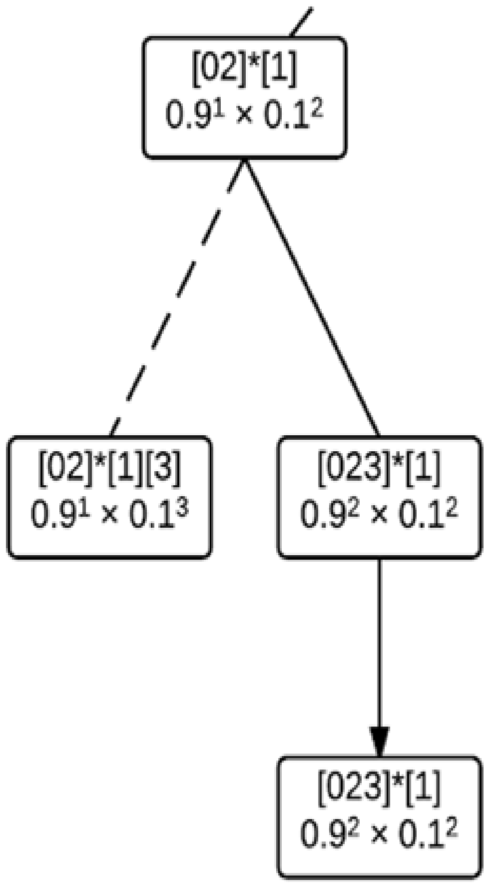


Reliability found



Reliability of source to K = $(0.9^2 \times 0.1^2) + 3(0.9^3 \times 0.1) + (0.9^4) = 0.8829$

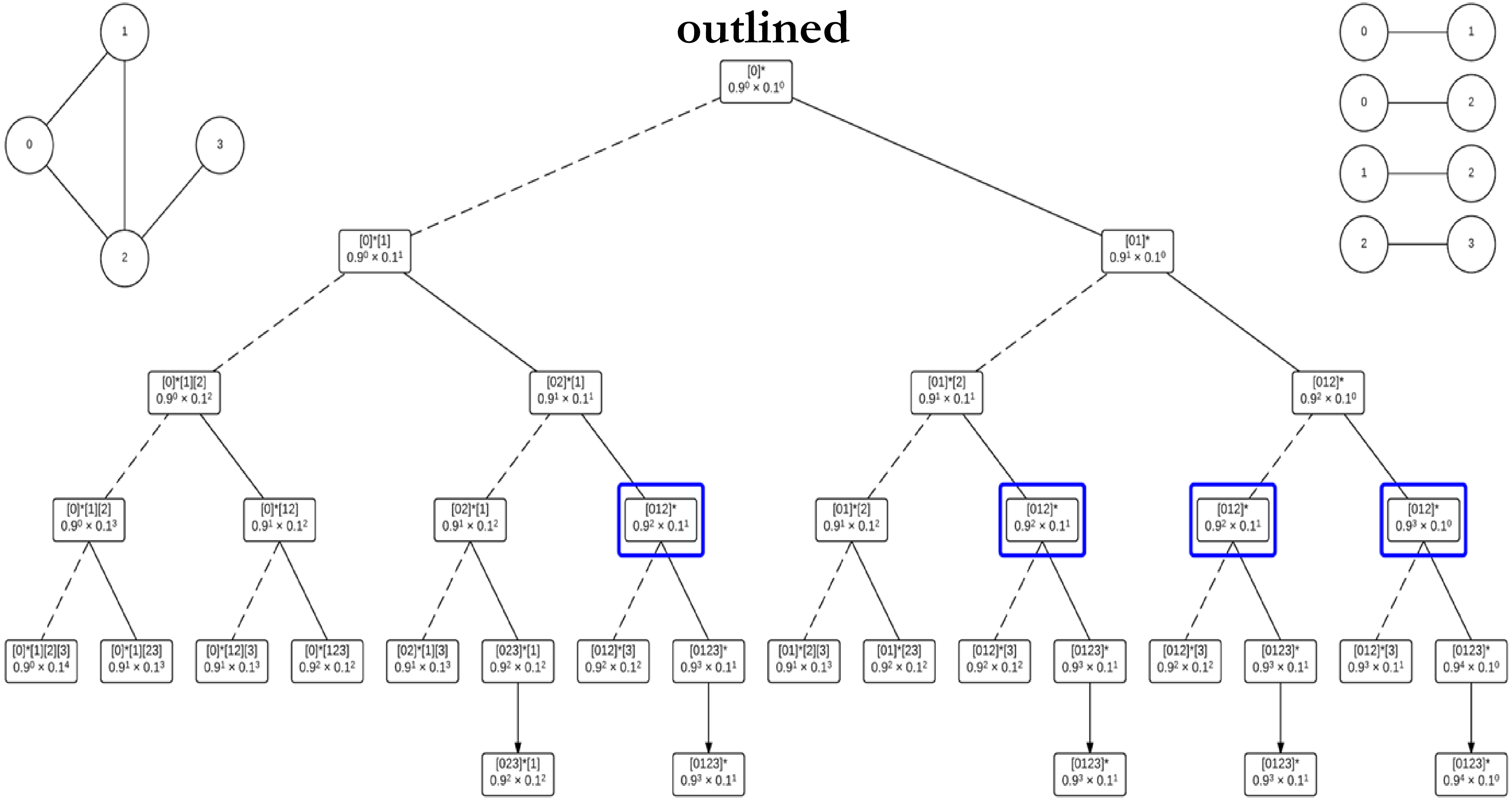
Storage of permutation information



Algorithm Efficiency

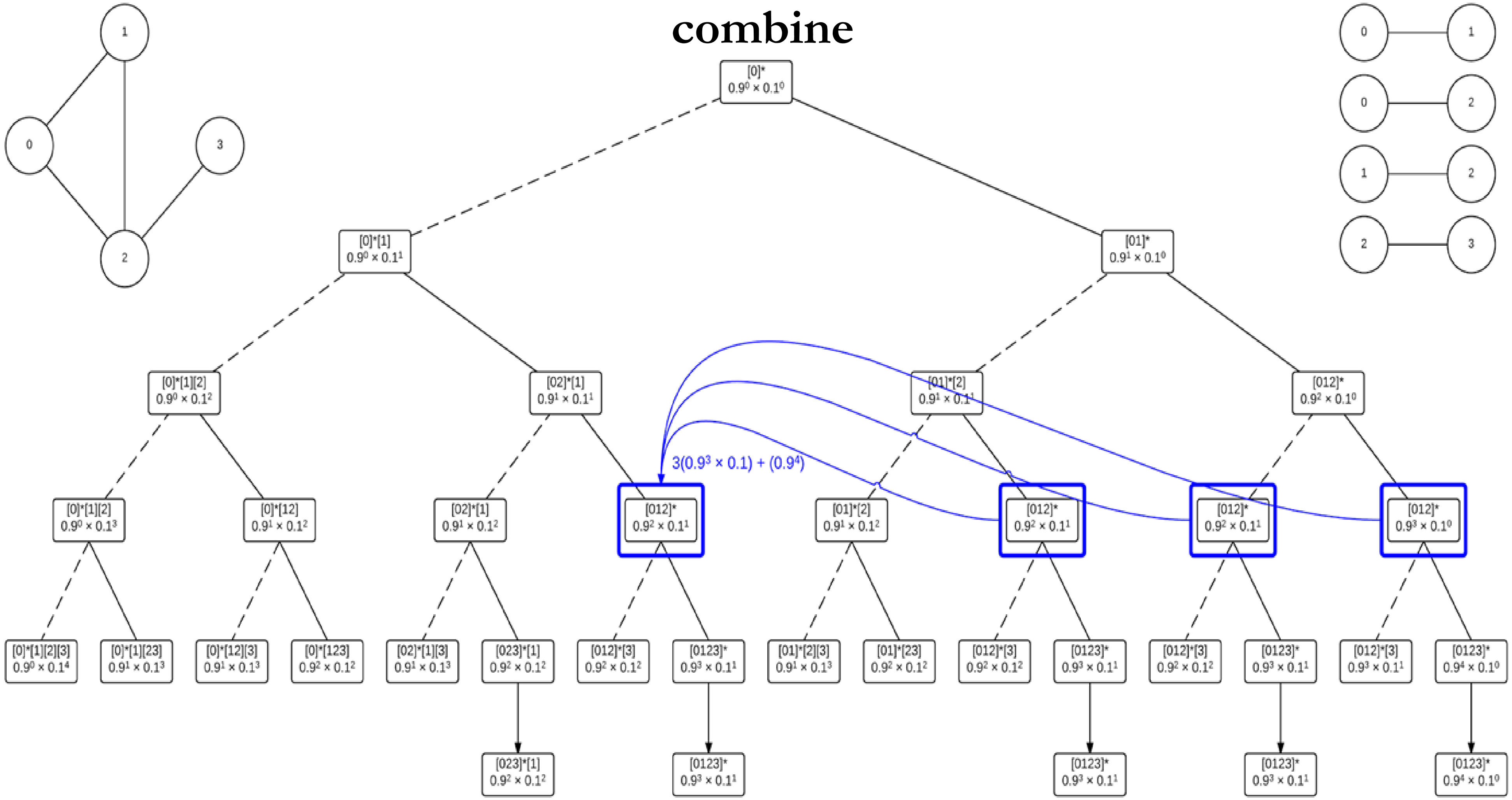
- Techniques used to reduce sub-problems
  - Isomorphism adds probabilities of similar subproblems
  - Redundant vertices are ignored to shorten the boundary set
  - Terminal nodes are removed

Isomorphic nodes outlined



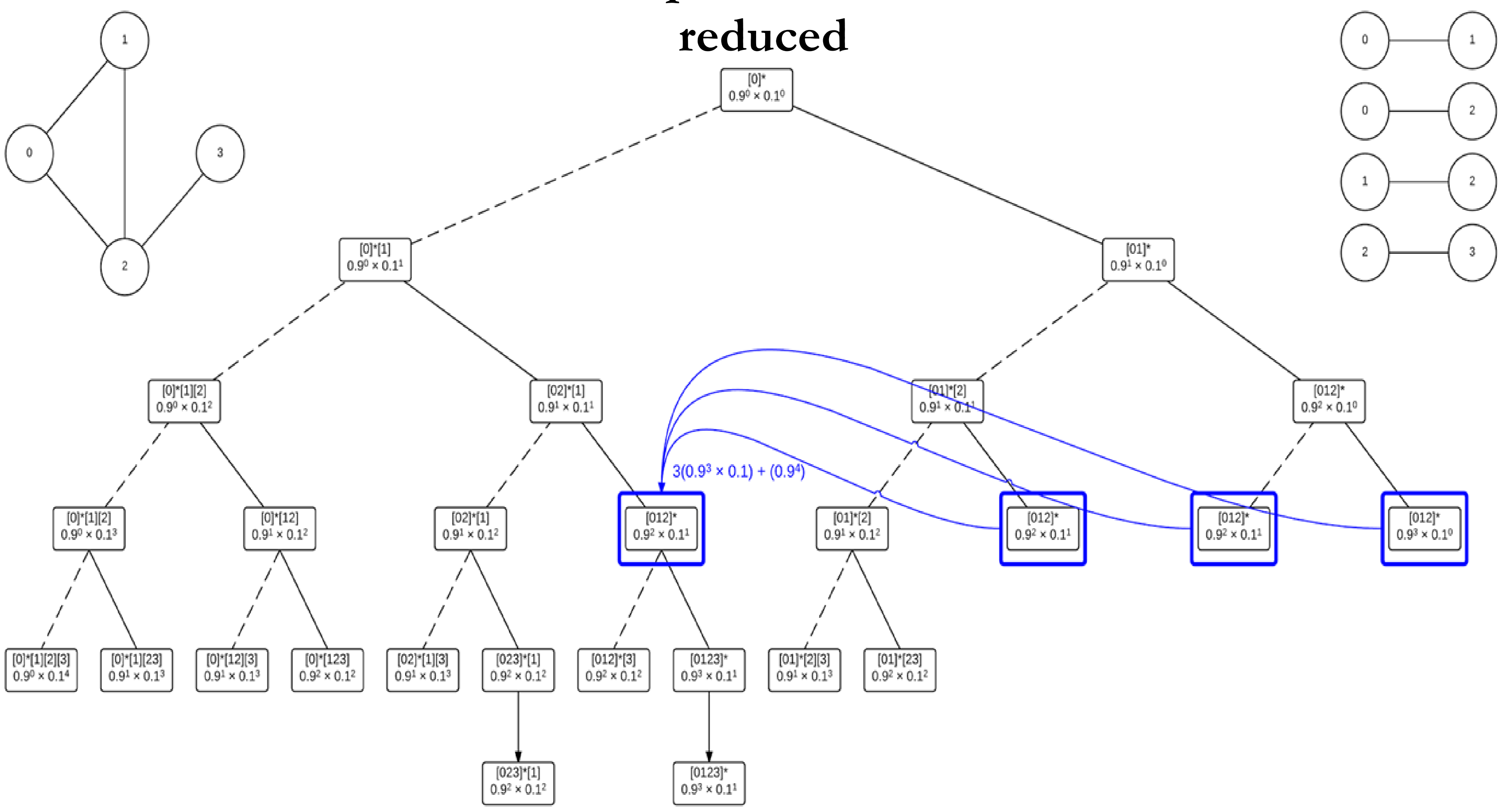
Reliability of source to K = $(0.9^2 \times 0.1^2) + 3(0.9^3 \times 0.1) + (0.9^4) = 0.8829$

Probabilities combine



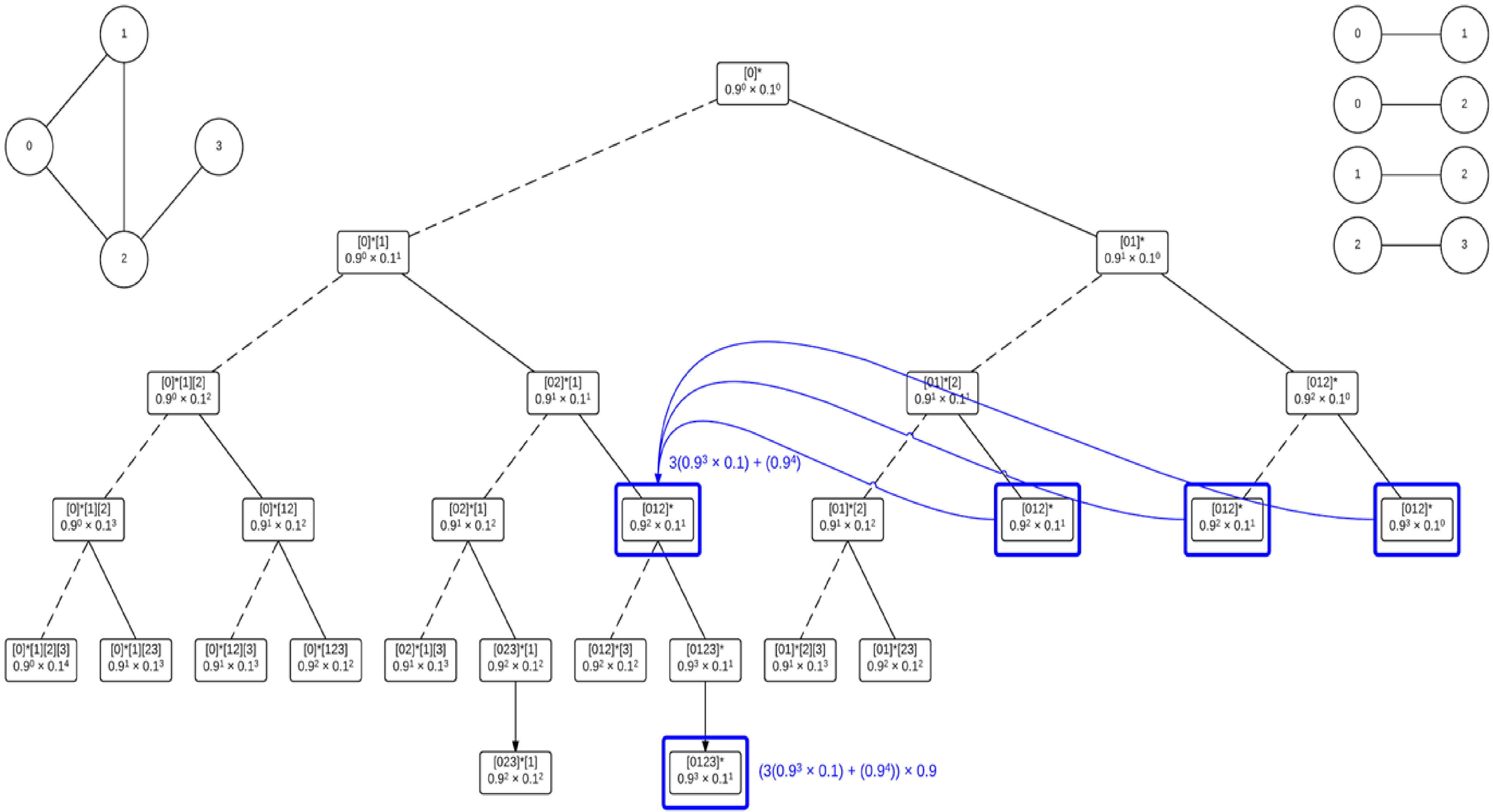
Reliability of source to K = $(0.9^{2} \times 0.1^{2}) + 3(0.9^{3} \times 0.1) + (0.9^{4}) = 0.8829$

Subproblems are reduced



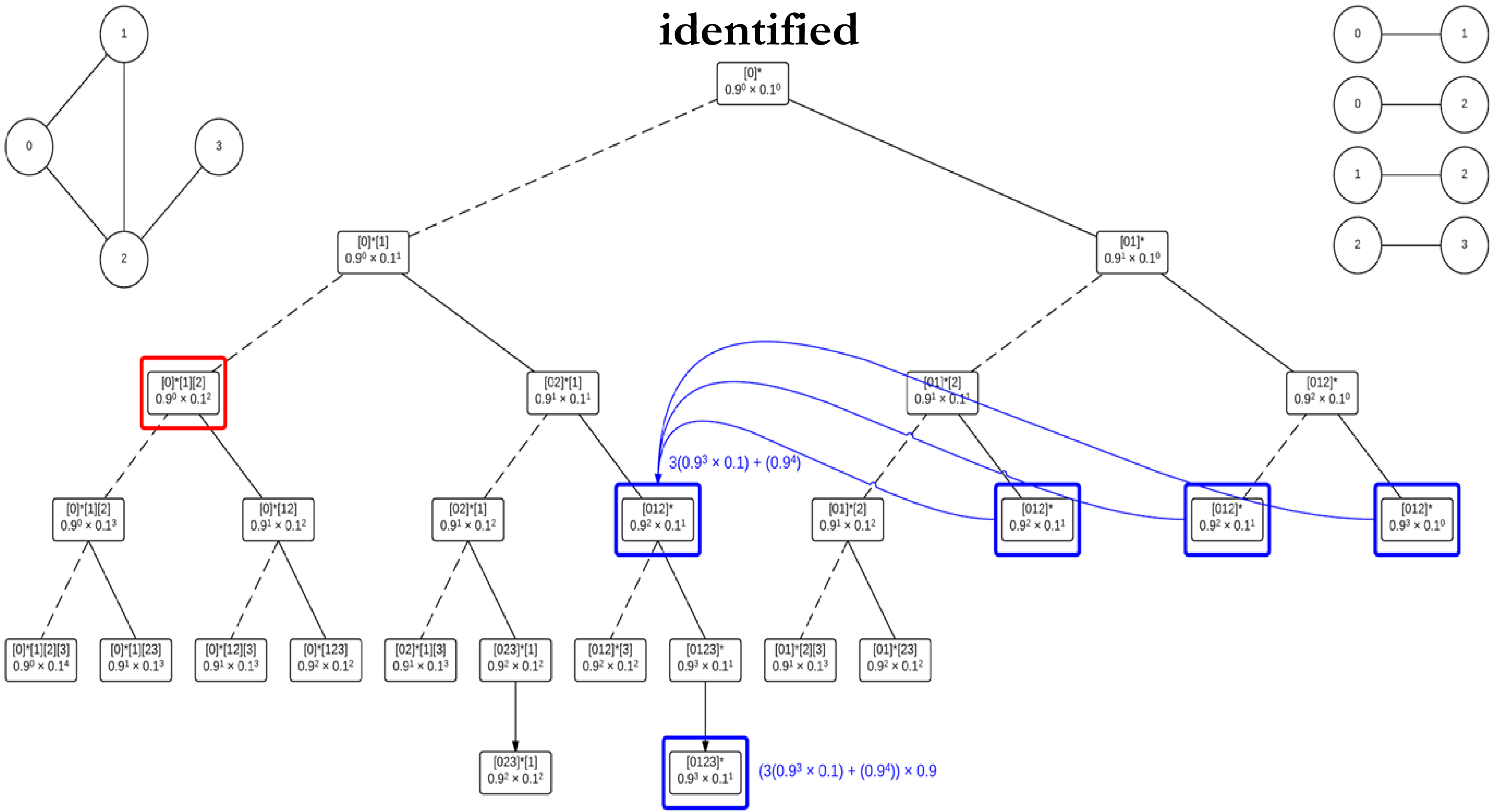
Reliability of source to K = $(0.9^2 \times 0.1^2) + 3(0.9^3 \times 0.1) + (0.9^4) = 0.8829$

Result is preserved



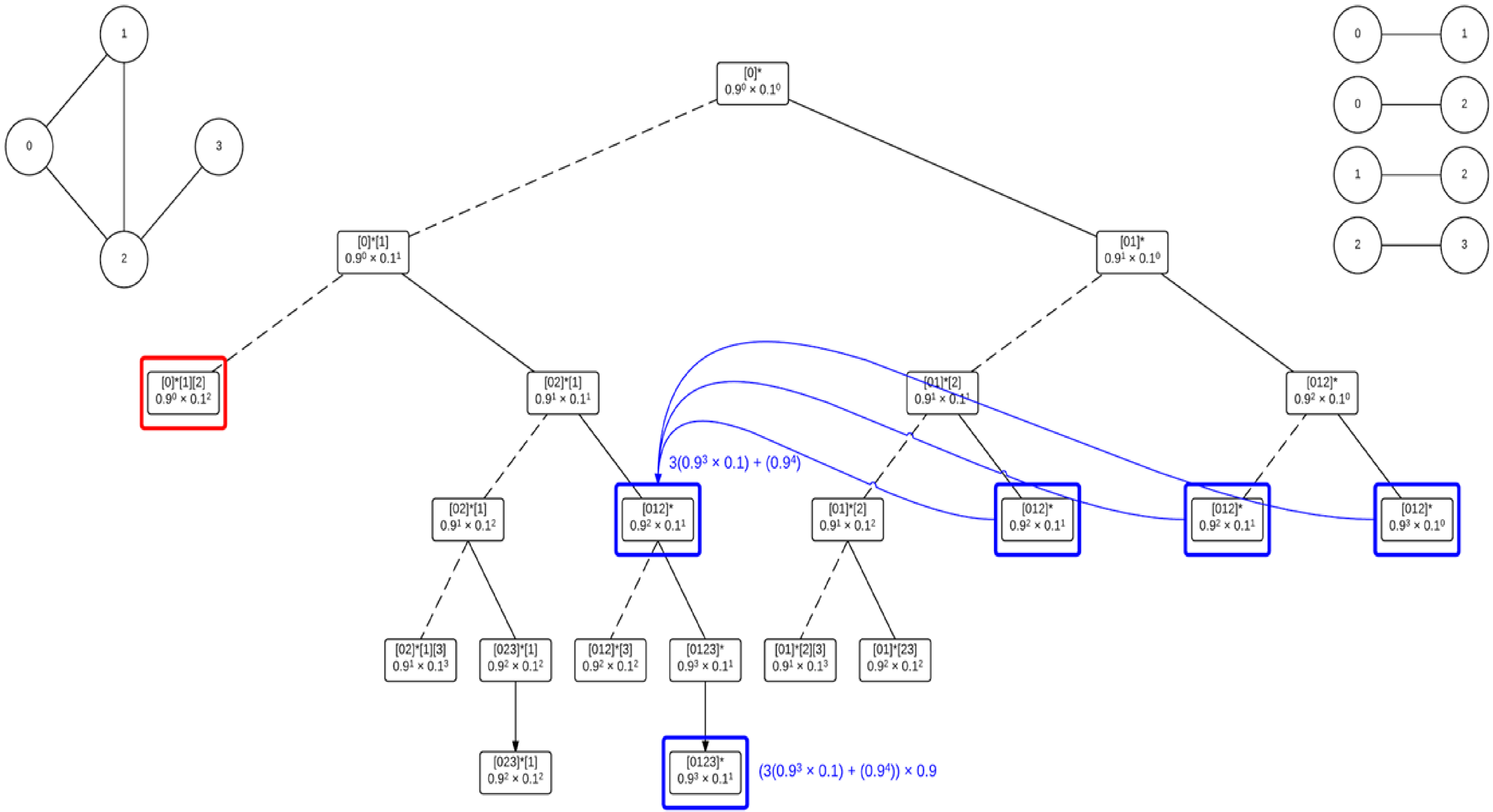
Reliability of source to K = $(0.9^2 \times 0.1^2) + 3(0.9^3 \times 0.1) + (0.9^4) = 0.8829$

Terminal node identified



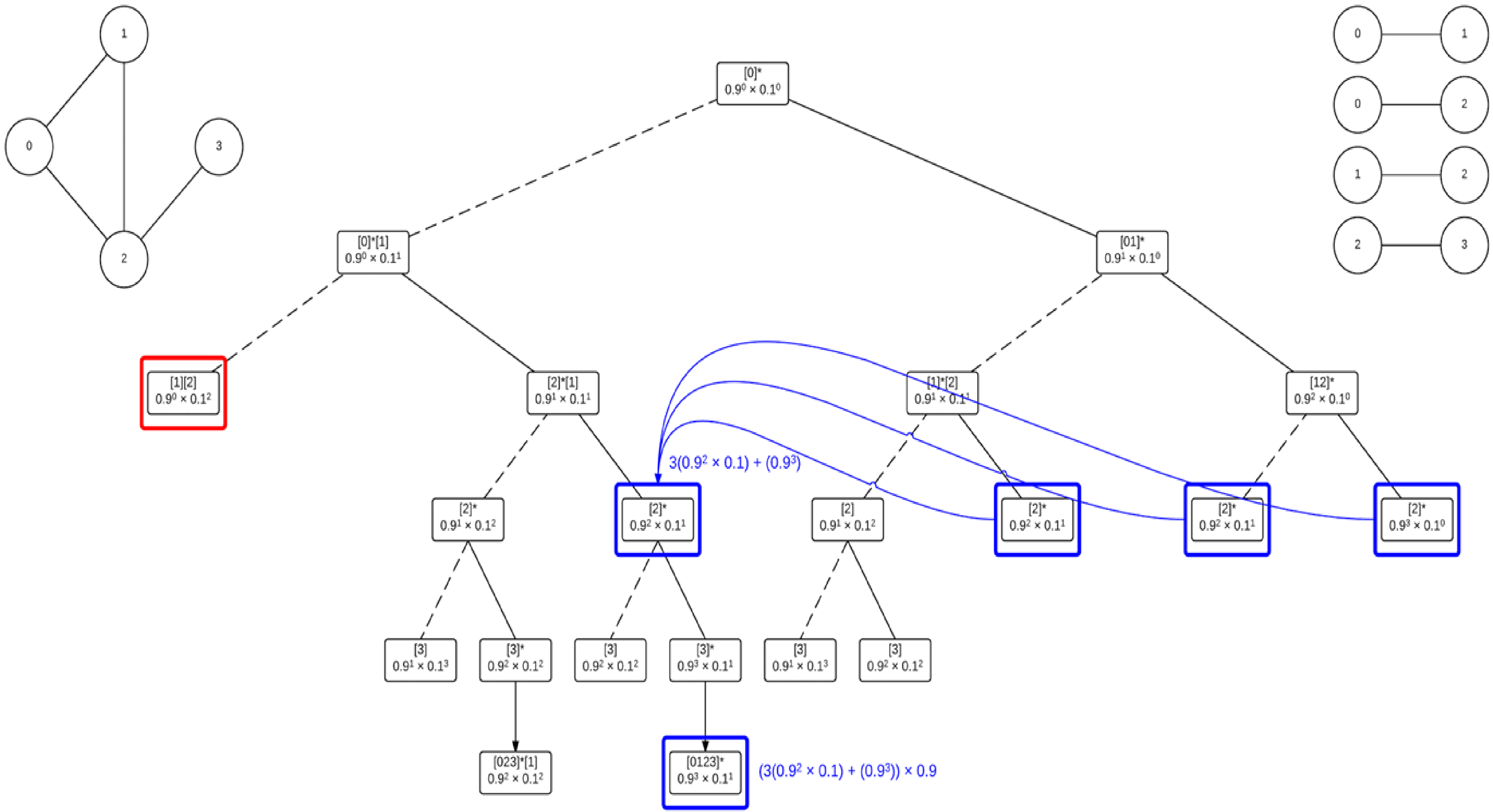
Reliability of source to K = $(0.9^2 \times 0.1^2) + 3(0.9^3 \times 0.1) + (0.9^4) = 0.8829$

Subproblem ignored



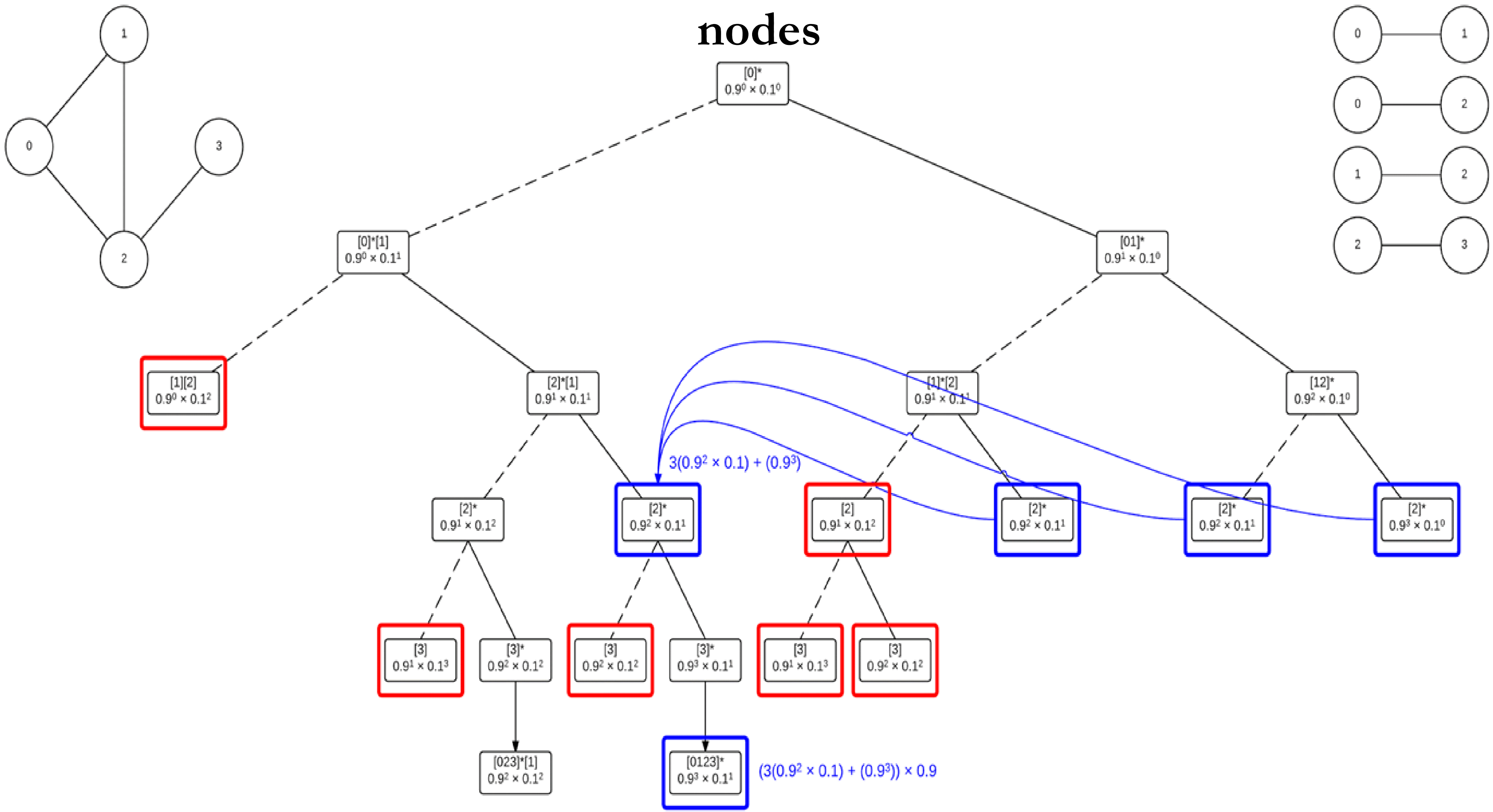
Reliability of source to K = $(0.9^2 \times 0.1^2) + 3(0.9^3 \times 0.1) + (0.9^4) = 0.8829$

Vertex reduction



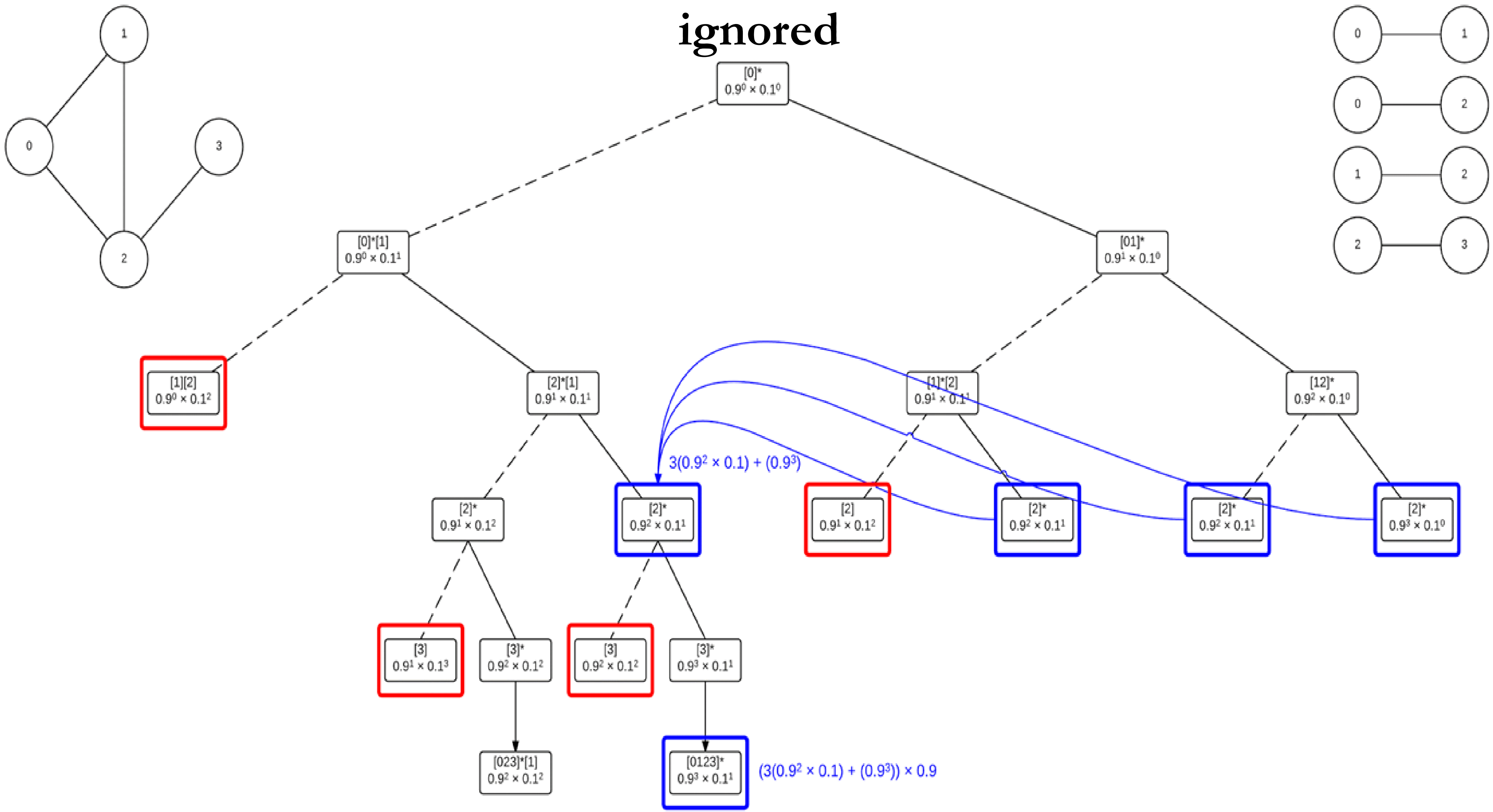
Reliability of source to K = $(0.9^2 \times 0.1^2) + 3(0.9^3 \times 0.1) + (0.9^4) = 0.8829$

Additional terminal nodes



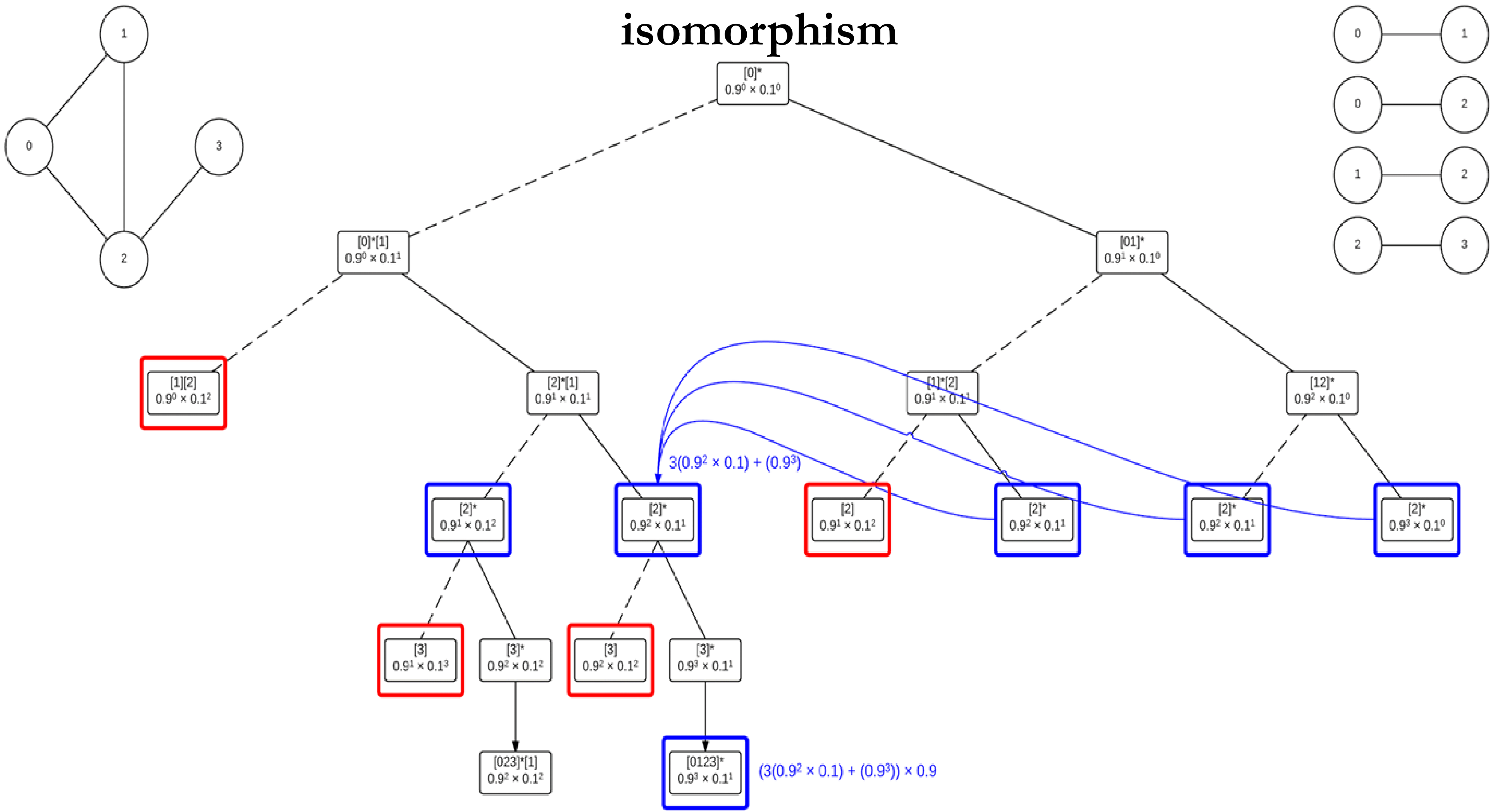
Reliability of source to K = $(0.9^2 \times 0.1^2) + 3(0.9^3 \times 0.1) + (0.9^4) = 0.8829$

Subproblems ignored



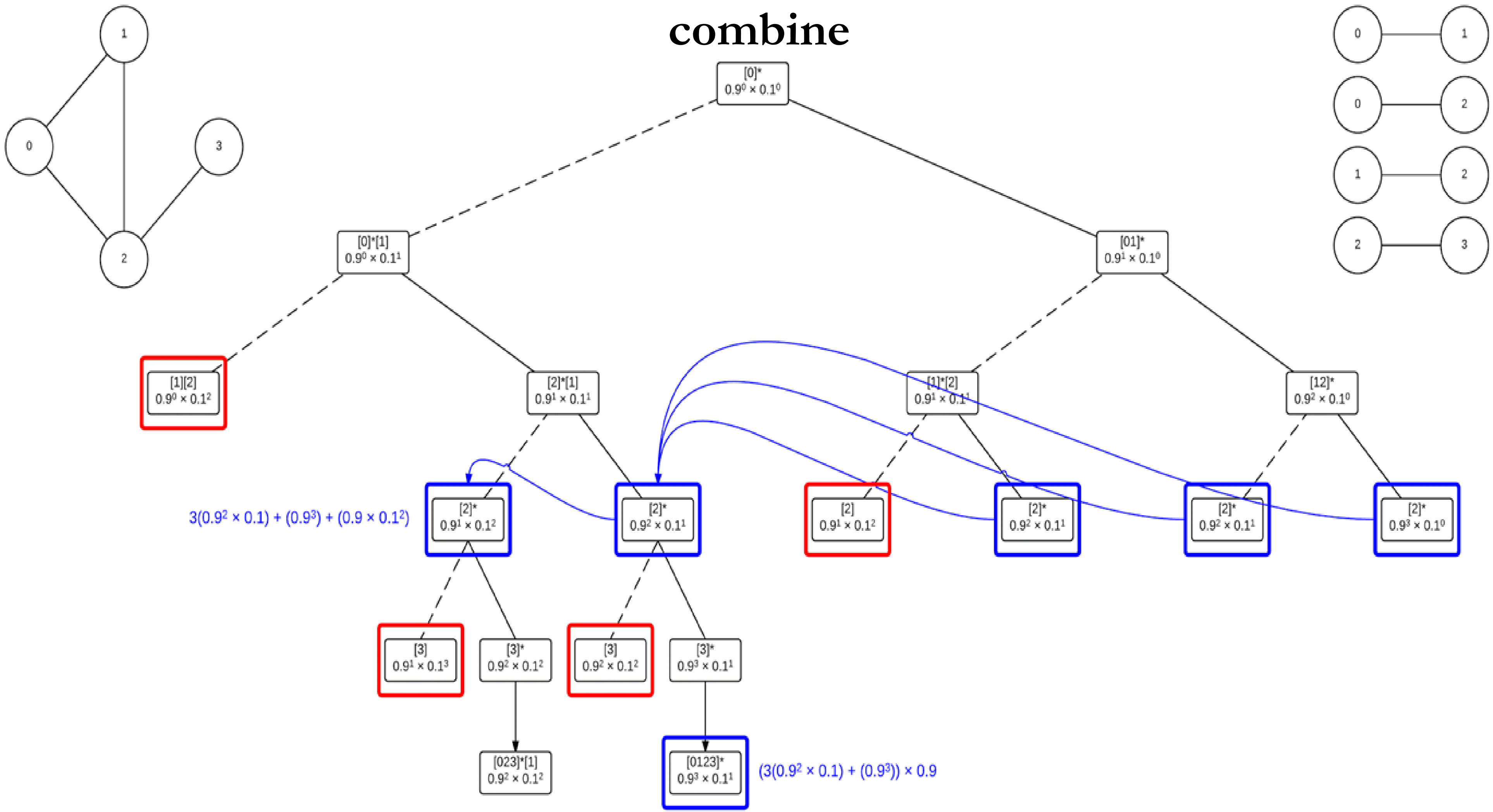
Reliability of source to K = $(0.9^2 \times 0.1^2) + 3(0.9^3 \times 0.1) + (0.9^4) = 0.8829$

Additional isomorphism



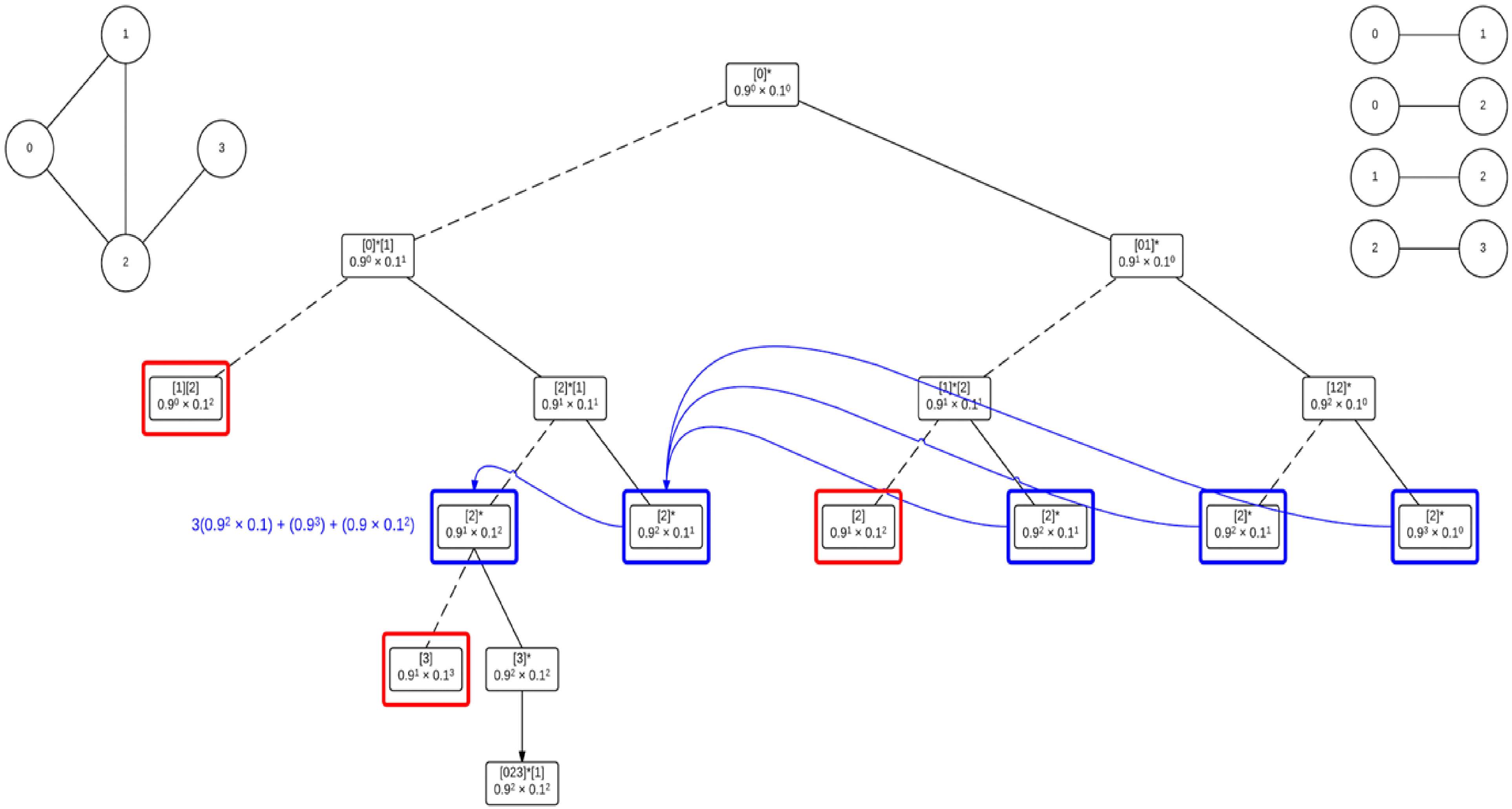
Reliability of source to K = $(0.9^{2} \times 0.1^{2}) + 3(0.9^{3} \times 0.1) + (0.9^{4}) = 0.8829$

Probabilities combine



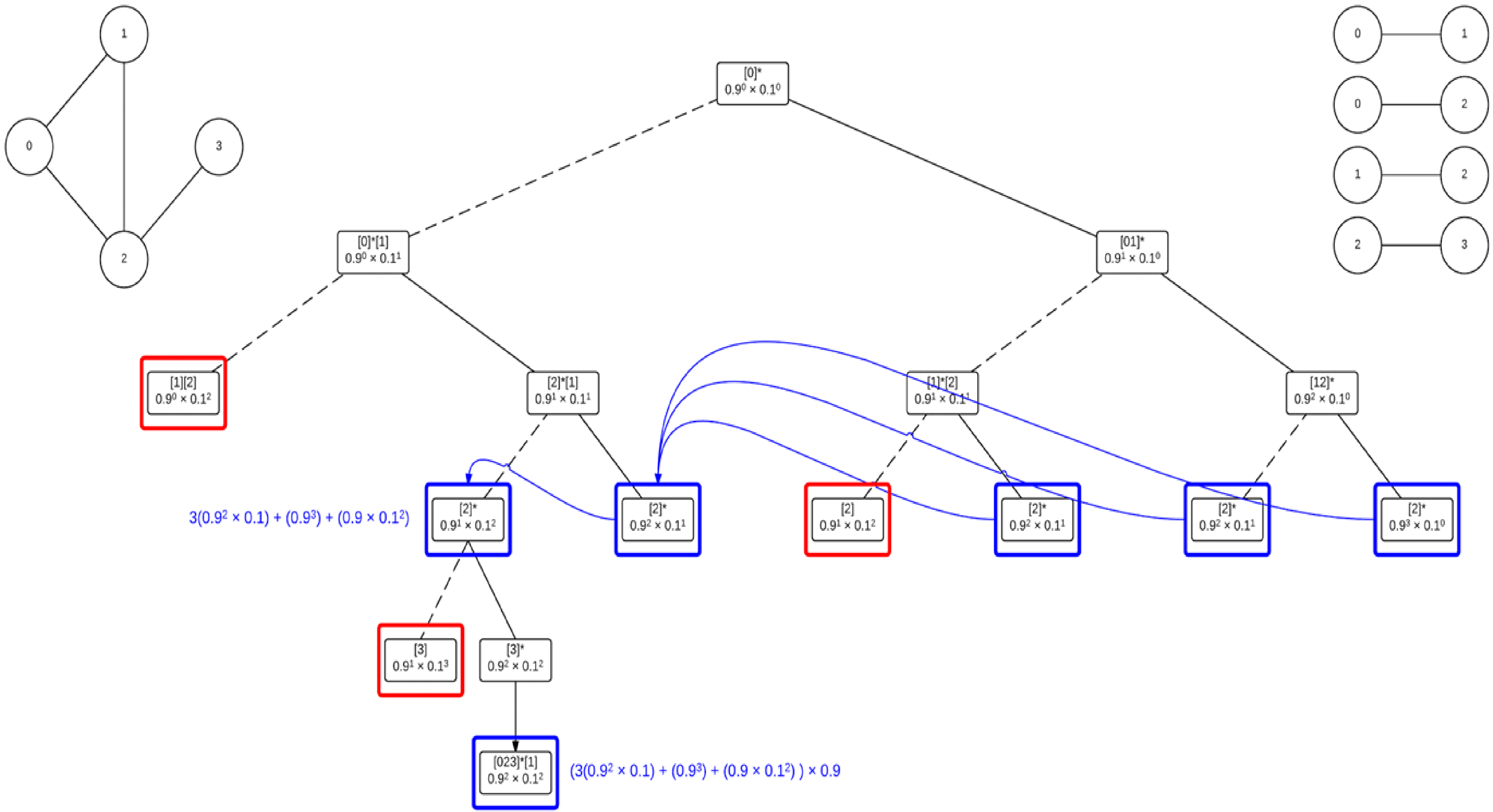
Reliability of source to K = $(0.9^2 \times 0.1^2) + 3(0.9^3 \times 0.1) + (0.9^4) = 0.8829$

Subproblem reduced



Reliability of source to K = $(0.9^2 \times 0.1^2) + 3(0.9^3 \times 0.1) + (0.9^4) = 0.8829$

Result is preserved



Reliability of source to K = $(0.9^2 \times 0.1^2) + 3(0.9^3 \times 0.1) + (0.9^4) = 0.8829$

The Actual End

– Listen to Captain Sparrow – this advice will come in handy during your degree.