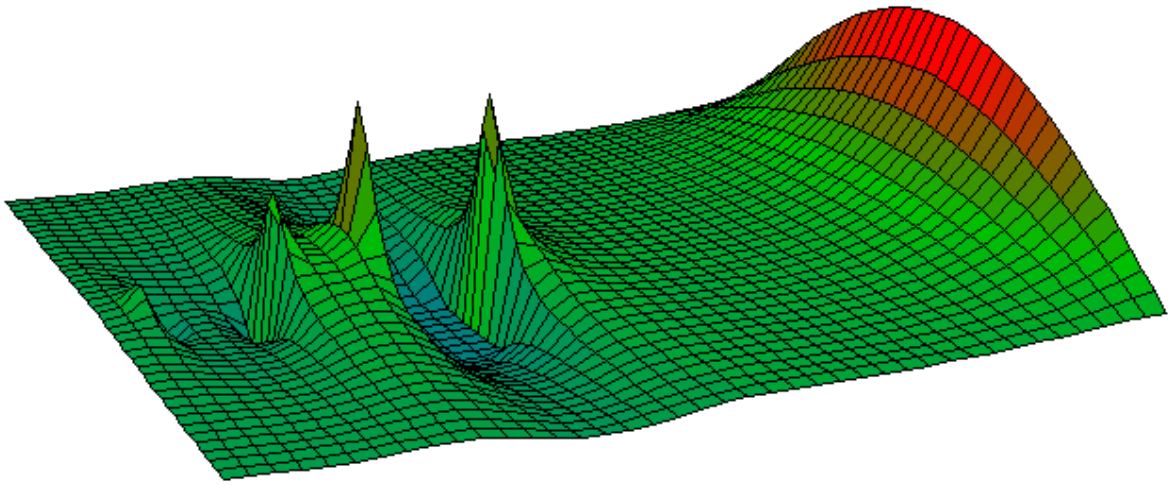
Journal of J

An interdisciplinary journal on J programming language and applications in science and technology.

New web site www.journalofj.com



Please Circulate

MPM press

ISSN: 2174-9280

An open access Journal

Vol.2, No.2 October 2013

JoJ (J2-team)

J, a language for the science, technology and more…

Aims and scope

Journal of J is a **not for profit** journal , involving a large research and users community in J programming language.

Journal of J is an interdisciplinary **journal** devoted to J and science and technology.

Journal of J aims to provide fast access to papers about science, technology and J.

Journal of J embodies the following principle:

Open Access: Knowledge is a public good. All readers have open access to reading and downloading papers. The simple and free access ensures maximum readership and high citation records for published papers.

Contact: journalofj at hotmail dot com (CONTRIBUTIONS)

info at journalofj dot com (GENERAL)

mikelpater at hotmail dot es (EDITOR)

Style and Contents

Journal of J aims to cover all the main areas of science. Inevitably, articles in different areas are addressed at different audiences. Many of the articles submitted to the journal are standard technical pieces, addressed to a purely academic audience

To attract this variety of contributions ***Journal of J*** will contain the following areas:

- Mathematics: number theory, logic, calculus, algebra, arithmetic, algorithms and others…
- Physics: dynamical systems, chaos, fractals, disorder, statistical physics and others…
- Computer science, Visualization, Engineering, Computer Art, …
- **others about J and J applications**

Visit: www.journalofj.com **Edited by Mikel Paternain**

Editorial:

Thanks to J Software, authors in this issue and J community !!!!!!

---

mikelpater at hotmail dot es

Contents

Steganography – the J way

Martin Saurer

martin DOT saurer AT bluewin DOT ch

1. Introduction

Steganography is the art of hiding data within data, so no one can see or hear (in case of hidden messages in audio files) the secret message, except that persons who use the right piece of software to extract the hidden information. Today's media files such as bitmap images or audio or video clips are good carrier files because of their size, and their binary format which is not human-readable.

An old-school approach to hide secret information in natural data is the linguistic steganography: the art of hiding text in text.

According to some conspiracy theories, there are some hidden prophecies in the bible-text, which predict the future of the world and human kind.

The first recorded use of steganography was about 500 years ago. Johannes Trithemius' work "Steganographia" is a treatise on cryptography and steganography disguised as a book on magic [1].

PRESIDENT'S EMBARGO RULING
SHOULD HAVE IMMEDIATE NOTICE.
GRAVE SITUATION AFFECTING
INTERNATIONAL LAW. STATEMENT
FORESHADOWS RUIN OF MANY
NEUTRALS. YELLOW JOURNALS
UNIFYING NATIONAL EXCITEMENT
IMMENSELY.

PERSHINGSAILSFROMNYJUNEI

PERSHING SAILS FROM NY JUNE I

A cablegram that might have sent by a journalist/spy from the U.S. to Europe during World War I.

2. Bitmap files

Just three peppers.
Or is there more information behind the scene?

In this article, we concentrate on hiding text in bitmap images. Bitmap files are great carriers to include more information, than seen by the naked eye using an ordinary image viewer.

Some bitmap formats like TIFF or JPEG may contain so called metadata to hold additional information like GPS coordinates (well known feature on smart-phones), annotation texts, and more. So, why not using the metadata section of an image to hide secret messages?

Just, because everybody looks there, first of all. The "security by obscurity" - concept only works, if the hidden secrets are not eye-catching, at a first glance. Therefore, the other way to hide a text in a bitmap image is to manipulate the pixels itself. The pixel modification must be large enough to represent some text, and small enough to be invisible by prying eyes.

3. Limitations

In our example, we will use image files in Bitmap (BMP) format. We depend on the fact that each pixel contains a small but essential piece of information. Under no circumstances, pixels must be changed by resizing the image, modifying the colors or converting it to a lossy data compression format like JPEG, after the embedding of a secret text. Nevertheless, converting a BMP file to the lossless PNG format is fine, and will work also the other way around.

Keep these limitations in mind when uploading an image to a social network or photo sharing service. Some services convert images to JPEG or "enhance" them in another way, after uploading. Your secret message will be corrupted, and no longer readable.

Of course, there are ways to embed data into an image so it will survive most image manipulations. To do so, a suitable error correction algorithm must be implemented when embedding and extracting the hidden text. But this will make steganography highly complex and goes far beyond this article.

4. Let's do it

In J, there is a small but great add-on (graphics/bmp) to read and write BMP files on each platform (Windows, Mac, Linux). When reading a BMP file, using "`readbmp`" you will get a 2-dimensional array holding the RGB (Red, Green, Blue) values of each pixel.

```
load 'graphics/bmp'

]bmp =. readbmp jpath '~user/Steganography/Moon.bmp'

4018799 4084591 4150385 4084849 4085105 4084850 4281971 4413299 …
4149616 4150385 4150641 4085105 4281714 4084850 4216691 4216434 …
4346736 4085105 4019314 3953777 4281970 4085363 4151155 4216435 …
4084848 3888241 4085107 4150899 4151155 4019826 4216948 4282228 …
4019569 4019569 4085105 4151154 4085618 4151156 4216949 4282484 …
4282226 4216691 4348020 4216434 4282484 4281972 4216693 4282229 …
4085362 4151155 4282484 4348020 4348021 4413813 4348277 4479093 …
4019826 4019826 4216948 4282484 4216949 4348276 4282740 4610934 …
…

$bmp

100 200
```

With a little J-magic, it's very easy to extract the bit-planes, and convert our 2-dimensional array into a 3-dimensional one, holding each bit-plane in a separate array.

```
NB. Convert integer-image-array to rgb-image-array
```

```
NB. Usage: int2rgb <integer-image-array>
int2rgb =: 3 :
'|:>(|:(<.256|(<.(<.y%256)%256)));(|:(<.256|(<.y%256)));(|:(<.256|y))'

rgb =. int2rgb bmp

$rgb

100 200 3
```

```
]red_bit_plane =. |:0{|:rgb
 61 62  63  62  62  62  65  67 …
 63 63  63  62  65  62  64  64 …
 66 62  61  60
 62 59  62  63
 61 61  62  63
 65 64  66  64
 62 63  65  66
 61 61  64  65
…
```

```
]green_bit_plane =. |:1{|:rgb
 82  83  84  84
 81  84  85  85
 83  85  84  84
 84  84  85  86
 85  85  85  87
 87  87  88  86
 86  87  88  88
 86  86  88  88
…
```

```
]blue_bit_plane =. |:2{|:rgb
111 111 113 113 113 114 115 115 …
112 113 113 113 114 114 115 114 …
112 113 114 113 114 115 115 115 …
112 113 115 115 115 114 116 116 …
113 113 113 114 114 116 117 116 …
114 115 116 114 116 116 117 117 …
114 115 116 116 117 117 117 117 …
114 114 116 116 117 116 116 118 …
…
```

All bit-planes (red, green, blue)

Red bit-plane

Green bit-plane

Blue bit-plane

To keep things as simple as possible, we use only the blue bit-plane to hide our secret message. The blue bit-plane contains the least significant bits, and we will use only one bit per pixel. So this method is also called LSB-Steganography (Least Significant Bit - Steganography).

The next step is to convert our 2-dimensional blue bit-plane array into a 1-dimensional array, so it looks like a simple byte stream.

```
]bbs =. ,/bpl

111 111 113 113 113 114 115 115 114 114 115 117 116 118 117 118 119 …
```

As you can see easily, there are even and odd numbers. If we replace all even numbers by 0 (zero) and all odd numbers by 1 (one), our byte stream will look like this:

```
111 111 113 113 113 114 115 115 114 114 115 117 116 118 117 118 119 …
  1   1   1   1   1   0   1   1   0   0   1   1   0   0   1   0   1 …
```

The resulting random bit stream is unusable to hide a text, so let's convert all these values to even numbers, by changing the least significant bit:

```
]bbse =. bbs - 2|bbs

110 110 112 112 112 114 114 114 114 114 114 116 116 118 116 118 118 …
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 …
```

Our blue bit-plane is now ready to receive the secret text. As an example, we take a quote from the famous TV-Series "The X-Files": "The Truth Is Out There". In a first step, we convert the letters to their corresponding ASCII codes (_ means blank / ASCII 32):

```
a. i. 'The Truth Is Out There'

  T   h   e   _   T   r   u   t   h   _   I   s   _   O   u   t   _   T …
 84 104 101  32  84 114 117 116 104  32  73 115  32  79 117 116  32  84 …
```

Then, let's convert the ASCII codes into a bit stream (ensure 8 bits per byte):

```
,/#: a. i. 'The Truth Is Out There',(128{a.)

T               h               e               _               …
84              104             101             32              …
0 1 0 1 0 1 0 0 0 1 1 0 1 0 0 0 0 1 1 0 0 1 0 1 0 0 1 0 0 0 0 0 …
```

And finally, let's add the bits in secret message bit stream to our blue bit-plane (that one with only even numbers), and we will get:

```
  110 110 112 112 112 114 114 114 114 114 114 116 116 118 116 118 118 …
+   0   1   0   1   0   1   0   0   0   1   1   0   1   0   0   0   0 …
= 111 111 113 114 113 115 116 115 114 115 115 118 116 119 117 118 119 …
```

All we have to do now is to glue our bit planes together again, to produce a new bitmap image that contains our secret message:

```
NB. Convert rgb-image-array to integer-image-array
NB. Usage: rgb2int <integer-image-array>
rgb2int =: 3 : '<.((((|:0{|:y)*2^16)+((|:1{|:y)*2^8)+(|:2{|:y))'

bmp =. rgb2int |:>(|:rpl);(|:gpl);(|:bpl)
bmp writebmp jpath '~user/Steganography/Moon_new.bmp'
```

And that's all, so far.

5. Further explanations

When comparing the new blue bit plane to the original one, you will notice some differences:

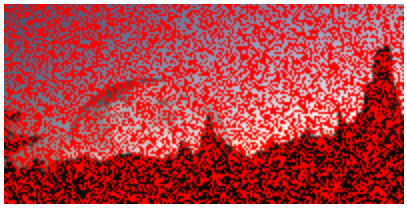
```
Old:  111 111 113 113 113 114 115 115 114 114 115 117 116 118 117 118 119
…
New:  111 111 113 114 113 115 116 115 114 115 115 118 116 119 117 118 119
…
Diff:             X       X   X           X       X       X
…
```

These differences are not recognizable by the naked eye, when using an ordinary image viewer. So the bitmap image with the hidden text is "nearly" the same as the bitmap without our secret message.

When analyzing both bitmaps at data level, you will recognize the differences easily:

Original Bitmap-Image

Bitmap-Image with embedded text



Differences, colored in red

What does that mean for the prying eyes of a secret service?

Keep your original bitmap image on your own computer, and never ever upload it to a cloud service, send it by email or make it available on the internet in any other way. Without the original bitmap image, it's much more difficult for code breakers, to find your hidden text.

Why did we have split up the three bit planes, when we just need the least significant bit?

Just for demonstration purposes. In this example, we only used the blue bit plane. But you may use all three bit planes to hide your text, by splitting the plain text in three parts, for example. Or you may hide three times more data, when using all of the three bit planes instead of only using one.

"Security by Obscurity" is a weak concept for storing secrets. What can I do, to make steganography more secure?

Pre-encrypt your secret text, before embedding it. It's much more difficult to detect a hidden message, when it's not directly machine-readable. 100% - security is an illusion! But you can make it more or less difficult for code breakers, to find and read your secret texts.

6. Steganography – the J way (full source code)

```
NB. Required scripts
require 'graphics/bmp'

NB. Utility verbs ***********************************************************

NB. Convert string to bit-stream
NB. Usage: <length> str2bit '<string>'
str2bit =: 4 : ',/_1}.#:((#y),(a.i.y),((0>.((x-1)-#y))$255?255),128)'

NB. Convert bit-stream to string
NB. Usage: bit2str <bit-stream>
bit2str =: 3 : '(#.(1+i.#.{.(((8%~#y),8)$y)){(((8%~#y),8)$y)){a.'

NB. Convert rgb-image-array to integer-image-array
NB. Usage: rgb2int <integer-image-array>
rgb2int =: 3 : '<.((((|:0{|:y)*2^16)+((|:1{|:y)*2^8)+(|:2{|:y))'

NB. Convert integer-image-array to rgb-image-array
NB. Usage: int2rgb <integer-image-array>
int2rgb =: 3 :
'|:>(|:(<.256|(<.(<.y%256)%256)));(|:(<.256|(<.y%256)));(|:(<.256|y))'

NB. Arc4 algorithm (based on Kym Farnik's work) *****************************

NB. Bitwise XOR
xor =: 22 b.
NB. Convert integer array to string
i2s =: 3 : 'y { a.'

NB. Convert string to integer array
s2i =: 3 : 'a. i. y'
NB. x = Key to use for encrypting and decrypting
NB. y = Plain Text or Encypted Text
NB.
NB. x can also be an integer array with integers >= 0 and <= 255
```

```
NB. with a maximum length of 256 elements
crypt =: 4 : 0
   st =. i. 256                          NB. list of 256 integers 0..255
   s2 =. x                               NB. Get key number list or string
   if. ' ' = {.0$s2 do.                  NB. If type is string then
      s2 =. s2i s2                       NB. make into list of numbers
   end.
   s2 =. 256 $ s2                        NB. vector 256 of the reapeting key
   i  =. j =. 0                          NB. set i,j to zero
   while. i < 256 do.                    NB. Setup state vector
      j  =. 256 | j + (i{st) + i{s2      NB. j = (j+state[i]+S2[i]) % 256
      st =. ((i,j){st) (j,i)}st          NB. Swap state [i] and [j]
      I  =. >: i                         NB. Increment i
   end.
   i =. j =. 0                           NB. set i,j to zero
   a =. s2i y                            NB. string as integer list ASCII
   n =. 0                                NB. set counter to zero
   while. n < #a do.                     NB. Traverse string
      i  =. 256 | >: i                   NB. Increment i mod 256
      j  =. 256 | j + i{st               NB. next J (j = j+state[i]) % 256)
      sw =. (i,j){st                     NB. Bytes to swap
      st =. sw (j,i)}st                  NB. store in swap order
      a  =. (((256|+/sw){st) xor n{a) n}a NB. k XOR string element[n]
      n  =. >: n                         NB. Increment n
   end.

                                         NB. Result: en/de-crypted string
   i2s a                                 NB. Convert integers to ascii
)

NB. ***************************************************************************
NB. Steganography
NB. ***************************************************************************

NB. Hides a message in a bitmap-image
NB. Usage: '<Message-to-hide>';<Password> hidemsg '<Bitmap-image-file>'
hidemsg =: 4 : 0
   tex =. >0{x                                    NB. 1st x-arg: message
   pwd =. >1{x                                    NB. 2nd x-arg: password
   bmf =. jpath y                                 NB. Full path to bmp in y
   stf =. (_4}.bmf),'_steg',(_4{.bmf)             NB. Build new file name
   bmp =. readbmp bmf                             NB. Read orig bitmap file
   dim =. $bmp                                    NB. Get bitmap dimensions
   msg =. (*/dim)$(*/dim) str2bit (pwd crypt tex) NB. Encrypt => bit stream
   rgb =. int2rgb bmp                             NB. Ints to RGB triples
   rpl =. |:0{|:rgb                               NB. Extract red bit plane
   gpl =. |:1{|:rgb                               NB. Extract green bit plane
   bpl =. ,/|:2{|:rgb                             NB. Extract blue bit plane
   bpl =. dim $ ((bpl - 2|bpl) + msg)             NB. Embed encrypted text
   bmp =. rgb2int |:>(|:rpl);(|:gpl);(|:bpl)      NB. Create new bmp image
   bmp writebmp stf                               NB. Write new bitmap file
)

NB. Extract hidden message from bitmap-image
NB. Usage: <Password> showmsg '<Bitmap-image-file>'

showmsg =: 4 : 0
   pwd =. x                                       NB. Password in x-arg
   bmp =. readbmp jpath y                         NB. Bitmap file in y-arg
   rgb =. int2rgb bmp                             NB. Ints to RGB triples
   bpl =. 2|,/|:2{|:rgb                           NB. Blue bit plane (MOD 2)
   msg =. bit2str bpl                             NB. Bits stream => string
   tex =. pwd crypt msg                           NB. Decrypt => plain text)
)
```

J and NodeJS on the Local Machine

Zhe Hu

huzhe AT sigenics DOT com

1. Introduction

Nowadays modern web browsers are ubiquitously on both desktop and mobile computers. They are becoming the universal "virtual machines" that browser-based software can really "write once and run everywhere".

Using Javascript and helper libraries like JQuery and the accompanied HTTP technology such as AJAX and websocket, the browser window can provide matching user interface experience to the traditional desktop (native) application. But there is one shortcoming, that is the browser talks HTTP with a web server (in most cases remotely), not the operating system on the local machine directly.

For a "native" application that needs to access the local hardware (e.g. an Arduino board connecting to the USB) or the local file system, the obvious solution is to have a HTTP server running on the local machine at the same time.

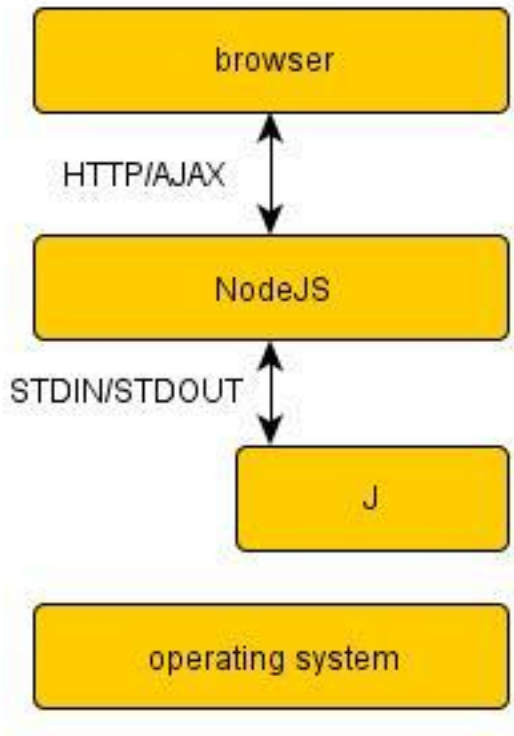
This local HTTP server coupled with a browser window GUI is exactly the architecture JHS adopts. In this article, we described an alternative architecture that uses NodeJS as the back-end HTTP server, instead of a J HTTP server (as shown in Fig. 1).

NodeJS is light-weight, but fast and powerful HTTP server software. To be more precise, you write the HTTP server in Javascript and NodeJS interprets your script on the fly, similar to jconsole interpreting J HTTP server code on the fly.

So how does J itself fit into the picture? J, as we know, has a very rich set of computational and algorithmic vocabularies. It's a perfect work horse for solving problems in the background.

The key is to "spawn" a jconsole as the child process of NodeJS while having NodeJS interface with the browser window. With the burden of serving HTTP shifted to NodeJS, J code can focus on doing what it is good at.

It does require J users to know Javascript to write an application this way. On the other hand, one needs to know Javascript anyway to be able to write the browser-based GUI code, even in the case of JHS.



Application Architecture

J as the Child Process of NodeJS

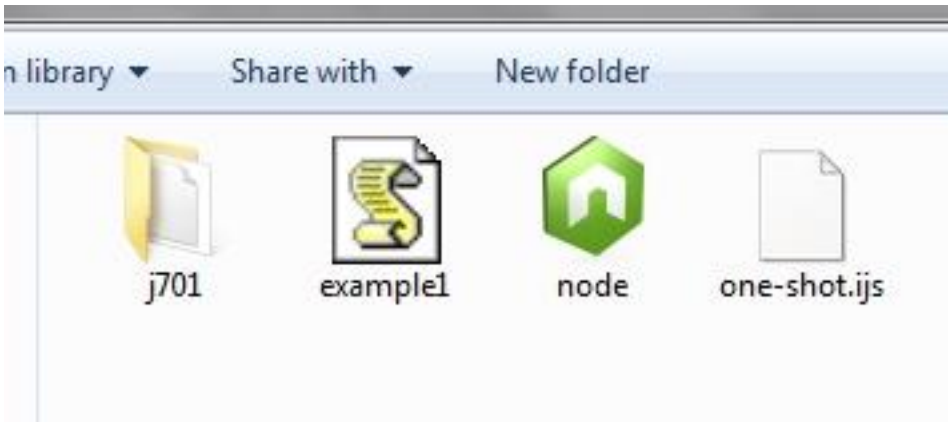
There are two ways for NodeJS to initiate jconsole and execute J code. The first method is called "one-shot". The Javascript code in NodeJS (example1.js) is shown below:

```
var exec = require('child_process').exec,
    child;

child = exec('j701\\bin\\jconsole one-shot.ijs 3 4',
  function (error, stdout, stderr) {
    console.log('stdout: ' + stdout);
    if (error !== null) {
      console.log('exec error: ' + error);
    }
});
```

The corresponding J code (one-shot.ijs) that reads 2 numbers from the command line input and adds them together:

```
'a b' =: 0".,.>2}.ARGV

myadd =: 3 : 0
  echo a + b
)
myadd''
exit''
```

It is assuming the following folder structure on a Windows machine (Fig. 2). The file path should be adjusted accordingly on other platforms.



Folder Structure

You can run the example program by typing in a windows console window:

```
node example1.js
```

The result is displayed in the console window

```
stdout: 7
```

Since it gets terminated by `exit''` at the end, `jconsole` returns control to the console window. It is certainly a very clean way to exit a child process in `NodeJS`. On the other hand, if every time `NodeJS` gets a HTTP request to calculate `myadd` in J, it needs to initiate J from ground-zero again. That could add up to the total response time.

The second method, called "keep-alive", spawns jconsole as a child process and keeps the stdin/stdout channel open (example2.js).

```
var spawn = require('child_process').spawn,
    jj    = spawn('j701\\bin\\jconsole', ['keep-alive.ijs']);

jj.stdout.on('data', function (data) {
  console.log('stdout: ' + data);
});


jj.stdin.write('3 myadd 4\n');

jj.stdin.write("exit''\n");
```

The corresponding J code (keep-alive.ijs)

```
myadd =: 4 : 'x + y'
```

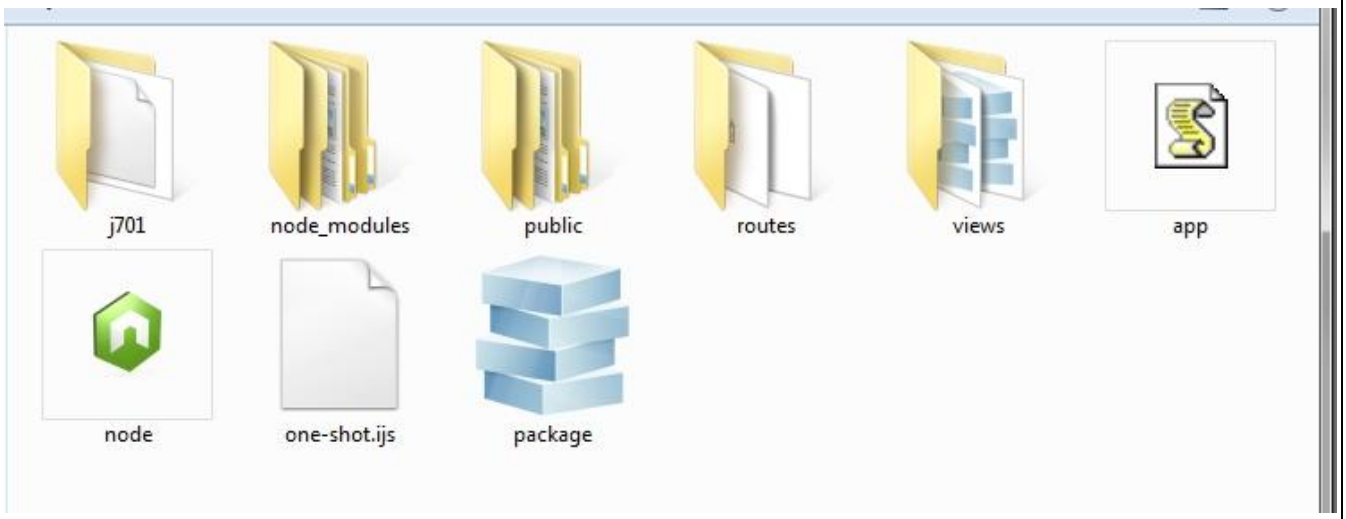
Notice that NodeJs needs to send jj.stdin.write("exit"\n"); to terminate jconsole eventually, otherwise jconsole becomes zombie.

NodeJS as HTTP Server

A simple HTTP server in NodeJS only takes a couple of lines of Javascript. But a more functional HTTP server needs to make use of additional Javascript libraries built on top of NodeJS. In this case, we choose ExpressJS. It is also very easy to scaffold a full-fledged web server in a folder "jnode", by typing

```
express jnode
```

Now you have all the necessary files in the "jnode" folder. You can copy the whole "j701" folder, as well as "node.exe" (for Windows) inside "jnode" to keep the application self-contained.



ExpressJS HTTP Server Folder Structure

In response to a "GET" request from the browser, we just need to add the following lines into the top level file "app.js".

```
var exec = require('child_process').exec;
app.get('/', function(req,res){
    exec('j701\\bin\\jconsole one-shot.ijs ' + req.query.a + " " +
req.query.b,
        function (error, stdout, stderr) {
            res.send("result: " + stdout);
        });
});
```

It's time to start the NodeJS HTTP server with:

```
node app
```

and input the following in the browser address bar:

```
http://localhost:3000/?a=3&b=4
```

You will get the sumed result in the browser window, which is calculated by J code (one-shot.ijs) at the back-end.

This is a very simple, but straight forward demonstration. It has the skeleton of a complete HTTP server with J code capability.

On the browser side, you can make HTML pages (e.g. via template engine) with forms to submit the GET request or to bind JQuery event function for AJAX, or even socketio between the browser and HTTP server. More advanced user could throw BackboneJS or other MVC framework into the browser side code as well.

Deployment

Let's come back to the objective of building browser-based software in the first place. It is because we want the code to be able to run in a wide variety of platforms. Let's check the components of our software so far.

- Our GUI software is Javascript code that runs in any modern browser.
- NodeJS backend runs on major desktop operating systems.
- J backend runs on major desktop operating systems.

Another nice thing about NodeJS and J is that installation process is simply copy-and-paste of folders. You can deploy the software to the end user with shared folder approach (Dropbox or Bittorrent Sync), which also makes updating the software very easy. Because of the ease of deployment, the application can be hosted on cloud platforms such as Amazon AWS, Windows Azure. Here is an sample application running on Windows Azure Website, with J code at the back-end: Class-E circuit calculator

Fibonacci Carpets

Cliff Reiter

Lafayette College, Department of Mathematics, Easton PA, 18042 U.S.A.

The Fibonacci numbers may be defined recursively by $F_0 = 0, F_1 = 1$, and $F_k = F_{k-1} + F_{k-2}$ for $k \geq 2$. This sequence is famous for extensive identities and many connections to nature and computation [1, 5, 6]. We explore a way of plotting a square with Fibonacci size edges and then putting squares with smaller Fibonacci numbers onto the available corners as "leaves". Iterating this process results in a remarkably full carpet. The pattern is motivated by the example in [5]; however, by automating the process we can carry the process further.

1. Two Examples

We use a variant of `f2a` from the Jsoftware wiki [3] to define a function giving the positive Fibonacci sequence.

```
   pos_fib_seq
3 : '{:|:+/\@|.^:(i.y) 0 1x'


   pos_fib_seq 5
1 1 2 3 5
```

We define a function `draw_fib_carpet` in the next section that creates a Fibonacci carpet to a specified level. Figure 1 shows the level 5 Fibonacci carpet. The purple square has edge length 5, the blue squares have edge length 3, the green squares have edge length 2, and the light green and red squares have edge length 1. The purple square has 4 leaves and the other squares along the diagonals have three leaves. However, while the green squares off diagonal have 3 leaves, two of them are "shared". A similar fact is true of some of the light green squares off the diagonals that are near the edge where they share red leaves. Note that the innermost light green squares have two leaves. If the light green square below the purple square is thought of as a leaf from the right, we would expect another leaf to the south-west; however, that would overdraw the green square. So when squares arise as leaves from two directions, we can only extend them in the intersection of the expected directions.

The fact that leaves at various levels have different sized lists of allowed leaf directions suggests that we should use boxed arrays for our lists of polygons and their allowed directions when implementing `draw_fib_carpet`.

In Figure 2 we see the level 6 Fibonacci carpet where the innermost yellow squares now have enough room to each have 3 red leaves. Thus, the reduced number of leaves noted above does not fully persist.

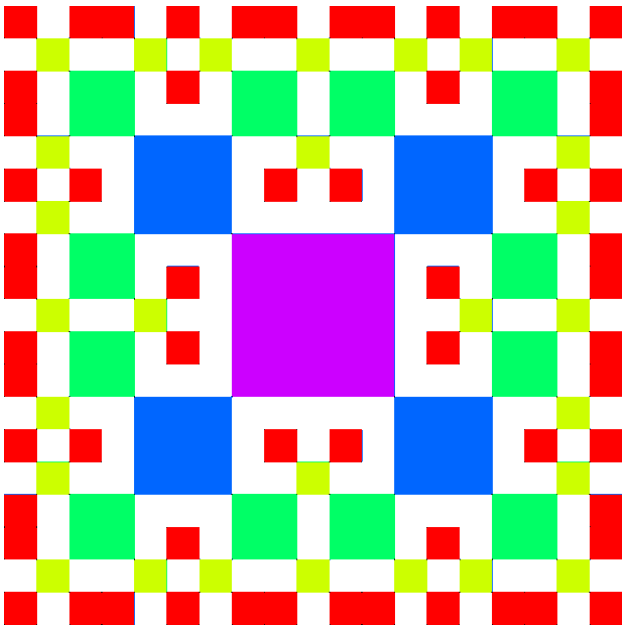


Figure 1*. The Fibonacci carpet at level 5 shows various numbers of leaves for various squares.*

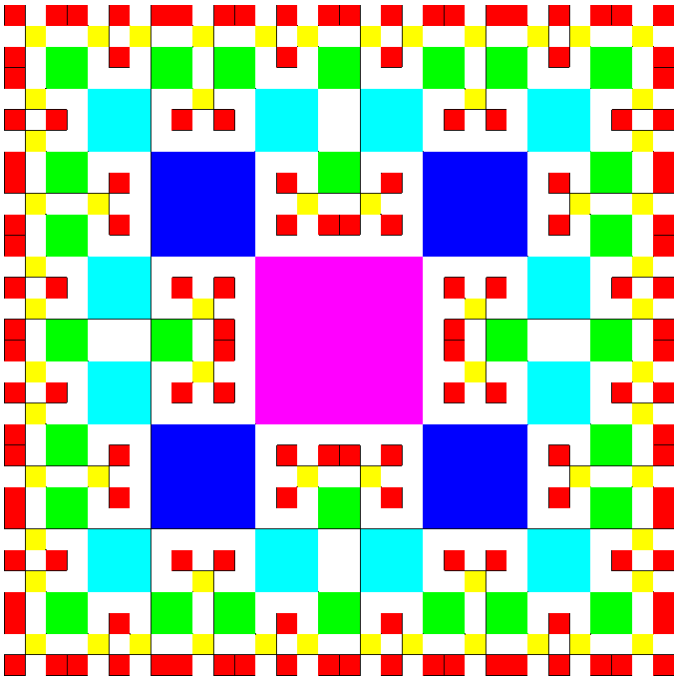


Figure 2*. The Fibonacci carpet at level 6 shows some of the inner squares have enough room to have 3 leaves again.*

2. The Implementation

The implementation is complex enough that we will not discuss the expressions in detail, but we will try to summarize the general idea behind each expression in `draw_fib_carpet`. First we note that

this implementation uses addons available for J6.02 32bit version [2]. A script containing the verb definition is available at [4].

The initialization phase of the `draw_fib_carpet` verb defines a color palette, `pal`, and a unit square `sq1`. The directions `dirs` give the four possible directions for leaves.

```
   ]sq1=.#:0 2 3 1
0 0
1 0
1 1
0 1
   ]dirs=.<sq1{_1 1
+-----+
|_1 _1|
| 1 _1|
| 1  1|
|_1  1|
+-----+
```

The Fibonacci sequence, `fibs`, is initialized and the initial level of polygons, `P0`, consists of a single square with edge length the largest element of fibs. The allowed directions for this square are all four; thus, `d1`, the directions allowed for `P0`, is all of `dirs`. The verb `ori` puts vertices of a square into a standard form (lower left, then counter clockwise). The verb `meet` computes the intersection of direction lists. The `range` is the coordinates of the lower left and upper right of the entire plot region. The extent, `ext`, is the lengths of the edges of that plot area. The global variable `WIN_WH` is set to be about 90% of what will fit on the screen. The verb `dwin` (defined by *dwin2.ijs*) creates a drawing window. The verb `dpoly` plots the polygon in `P0`. The verb `smoutput` is used to output the edge length and number of polygons plotted in the initialization phase.

In the iterative phase the leaf corners, `lc`, are computed and `P0` is updated to be the squares that are leaves moved into the directions allowed by `d1` and these are put into standard form. The directions in `d1` are updated for the current polygons by excluding the opposite of the direction used to produce the square. Then `d1` is updated again where the intersection of directions are taken for duplicate squares. Then the duplicate squares are removed from `P0`. The mask m is used to remove degenerate, "zero", polygons and the corresponding directions. Lastly, the polygons are plotted and the edge length and number of polygons is output. The commented expressions will be discussed in the next section.

```
load '~addons/graphics/fvj3/dwin2.ijs'
load '~addons/media/image3/view_m.ijs'
draw_fib_carpet=:3 : 0
pal=.Hue (i.%])y
sq1=.#:0 2 3 1
dirs=.<sq1{_1 1
fibs=.pos_fib_seq y
NB. ]fibs=.(-:>:%:5)^i. y
```

```
P0=.<sq1*_1{fibs
d1=.dirs
ori=.0 2 3 1&{@:/:~"2
meet=.[:,:~.@(,/)-.-./,-.~/
range=.2#_1 1*_2{.+/\fibs
ext=.(2 3{range)-0 1{range
WIN_WH=:x:^:_1 <.ext*<./0.9*(_2{.".wd'qm')%ext
NB. WIN_WH=:2000 2000
range dwin 'Fibonacci Carpet: level ',":y
(_1{pal)dpoly > P0
smoutput (_1{fibs),#P0
for_J. -2+i.y-1 do.
  lc=.(dirs i.&.> d1) {&.> P0
  P0=.,ori&.><"2 lc+"1"1 2&> (<sq1*J{fibs) *"1"_ 1&.> d1
  d1=.,<"2&> dirs -."_ 1&.> -&.> d1
  d1=., meet/`]@.(2>#)@:>&.> P0 </. d1
  P0=.~.P0
  m=.P0~:<4 2$0
  P0=.m#P0
  d1=.m#d1
  (J{pal)dpoly >P0
  smoutput (J{fibs),#P0
end.
NB. wd 'psel ',WIN_nam
NB. dfi=.glqpixels x:^:(_1)0 0,WIN_WH
NB. $draw_fib_image=:|.256 256 256#: (|.WIN_WH)$dfi
)
```

Figure 3 shows a 12 level Fibonacci carpet with the session log recorded below.

```
   draw_fib_carpet 12
144 1
89 4
55 12
34 28
21 60
13 132
8 300
5 692
3 1596
2 3668
1 8412
1 19284
```

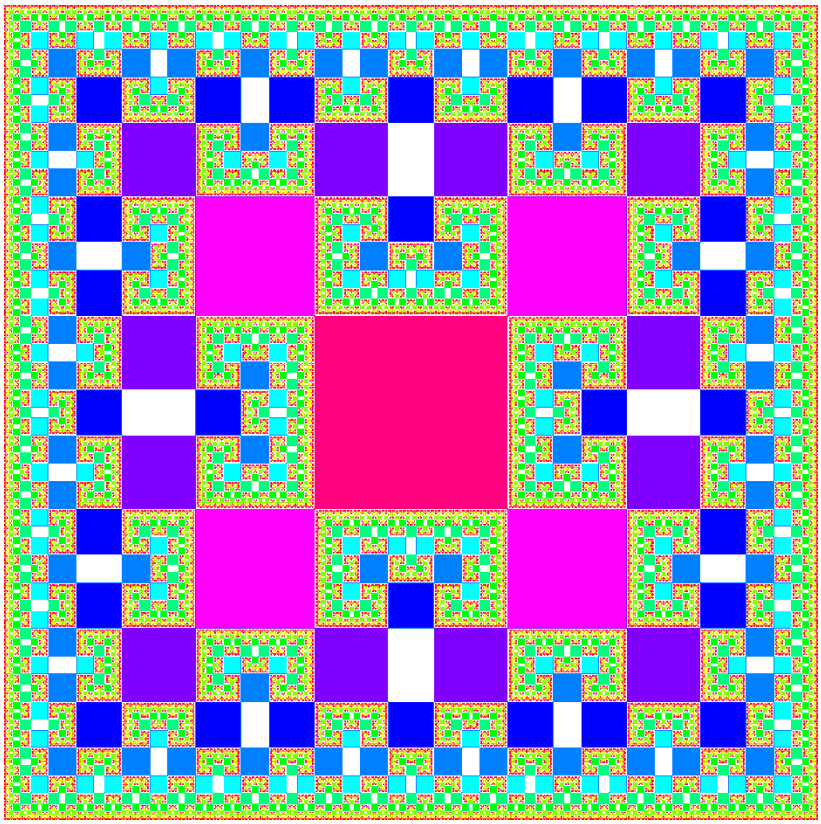


Figure 3. The Fibonacci carpet at level 12.

3. Variants

Instead of plotting squares with edge lengths given by the Fibonacci sequence we can plot squares with edge lengths given by the powers of the golden ratio. This can be accomplished by uncommenting the second line defining `fibs` in `draw_fib_carpet`. The result is often called the golden carpet (not shown). The result is very similar to the Fibonacci carpet but worth the experiment.

The Lucas sequence satisfies the same recursion as the Fibonacci sequence but using different initial conditions: $L_0 = 2, L_1 = 1,$ and $L_k = L_{k-1} + L_{k-2}$. This is gives another variation that produces a carpet

There are several additional commented lines in `draw_fib_carpet`. Uncommenting those lines (and using the original or variant carpet) allows high resolution raster versions, suitable for printing, to be created. In particular a global variable `draw_fib_image` is created that represents the image as an RGB raster image. The window width and height can exceed the screen resolution; it need not be the 2000 by 2000 suggested. A high resolution raster version of the image can then be saved via an expression similar to the following (use an appropriate path).

```
draw_fib_image write_image 'c:\fib_carpet.png'
```

4. Counting Squares

We do not have an analysis for computing the number of squares produced at each level of the Fibonacci carpet but these appear empirically to coincide with the following function.

```
   f=:,4+(2*{:)+2*+/@:(_3&}.)
   1,f^:(10) 4 12
1 4 12 28 60 132 300 692 1596 3668 8412 19284 44220
```

The ratio of the successive terms is slightly larger than 2.29 in the limit.

```
  18j15":x:^:_1]2 %~/\ _4{.1,f^:(100) 4 12
 2.293799982029938 2.293799982029938 2.293799982029938
```

In particular, this means the growth is exponential. One might hope that the ratio approaches a quadratic irrational, in which case it should have a periodic continued fraction expansion. Using a verb for converting a rational number into a continued fraction expansion, that does not appear to be the case.

```
   r_to_fcf=: 3 : 0
'r0 r1'=.2 x: y
z=.i.0
while. r1 ~: 0 do.
  q=.<. r0 % r1
  z=.z,q
  'r0 r1'=. r1,r0-q*r1
end.
z
)
   r_to_fcf %~/ _2{.1,f^:(60) 4 12x
2 3 2 2 10 2 12 1 1 3 3 80 20 2 1 2 1 16 1 2 5 2 1 1 14 1 2 1 11 1 1 2
4 13 12 2 14 4 5
```

References

[1] Richard A. Dunlap, *The Golden Ratio and Fibonacci Numbers*, World Scientific, 1997.

[2] Jsoftware, J6.02 (32 bit), with *media/image3* and *graphics/fvj3* addons,

http://www.jsoftware.com, 2008.[3] http://www.jsoftware.com/jwiki/Essays/Fibonacci Sequence

[4] Cliff Reiter, Fibonacci Carpet Script, *http://webbox.lafayette.edu/~reiterc/j/JoJ/fib_carpet.html*, 2013.

[5] Hans Walser, *The Golden Section*, translated by Peter Hilton et al, The Mathematical

Association of America, 2nd ed., 1996, translation, 2001.

[6] Wikipedia: Fibonacci Numbers, *http://en.wikipedia.org/wiki/Fibonacci_number*.

Elementary category theory of J

Viktor Z. Cerovski

Institute of Physics Belgrade,
Pregrevica 118, 11080 Belgrade, Serbia

27 Oct 2013

Abstract

J is described in terms of elementary category theory by identifying four classes of objects in J (nouns, verbs, adverbs and conjunctions) as objects and/or morphisms of a category. It is shown that in order to include all four classes into a categorical description of language it is necessary to consider several extensions of J, which are described in detail and shown to be completely orthogonal to the existing J. They open the way for both new programming techniques, which are briefly discussed, and new extensions (such as providing explicit constructs for parallel programming) fully compatible with the current J and oriented toward functional programming.

1 Introduction

Every J expression is defined in terms of four classes of objects: nouns, verbs, adverbs and conjunctions [1]. Typically, nouns stand for data, verbs operate on data to produce transformed data, adverbs modify verbs to become new verbs, and conjunctions link up two verbs or verb and noun to produce a new verb. Each verb can has its monadic and dyadic version, taking one or two noun arguments, respectively. Parsing of a J expression is carried out by the parsing machine described in Ref. [2].

Each of verbs, adverbs and conjunctions can be understood as a mapping. The goal of this paper to make this understanding precise by way of constructing a simple category theory of J.

Category theory has shown a great promise as a common mathematical foundation for a number of seemingly disparate areas: logic, topology, physics and computer science, to name a few [3]. In the context of computer science, the theory provides foundation for functional programming, where every object of a programming language is a constructive function [4].

A *category* **C** can be defined as consisting of *objects* together with *arrows* (also called *morphisms*.) An arrow connects a pair of objects, going from A to B. There is also an operation of making two successive connections, first by arrow g, then by f, of three objects (*composition* of morphisms), which is denoted by $f \circ g$. In addition, the following three conditions must be satisfied:

(1) For every object A, there is an arrow id_A, *identity arrow* of A, connecting A to itself. For every arrow f in **C** that connects A to B, it must hold

$f \circ id_A = f$, $id_B \circ f = f$.

(2) If an arrow f connects A to B, and arrow g connects B to C, then **C** contains also an arrow h connecting A and C defined by $h = g \circ f$.

(3) The composition of a sequence of connecting objects with arrows from **C** is associative:

$(f \circ g) \circ h = f \circ (g \circ h)$

An example of category are classical functions: each function *f* connects its domain to its codomain, which are sets; identity function provides identity arrow; functions can be composed to produce new functions; and the composition of functions is associative.

We begin the categorical exploration of J by considering the four classes of J objects in turn, and for each class consider whether it can be an object and/or an arrow of a category, how arrows compose and what is the identity morphism, and check for associativity of composition.

2 Nouns and their morphisms

Nouns are considered first, represented by the object *N* of **J**. Since every monadic verb maps noun to noun, verbs are natural candidates for arrows connecting *N* to *N*. Their composition is provided in J by the `conjunction @:` (At) [5]. Checking whether all this really satisfies the categorical laws (1-3) is done next.

Identity arrow for nouns, id_N in **J**, is in J simply the following verb

```
idN =: ]
```

because for every monadic verb `v` of J the following holds

```
idN @: v ≡ v ≡ v @: idN
```

where the relation ≡ means: left-hand side can be used in every expression where the right-hand side is used in its stead (and vice-versa) and the result of the replacement will be an expression that works in exactly the same way in J. Equality of two expression thus required is their referential transparency.

Thus, the law (1) has just been verified and] identified as the identity arrow. It is easy to see that] is not the only one, so, for instance, verb [will do just as well.

The law (2) is also satisfied since for any two built-in J verbs u and v, each acting on noun to produce noun, u@:v is also a valid verb in J acting on noun to produce noun.

The associativity law (3) can also be proved to be true. Indeed, for each three J verbs, the following

```
(u @: v) @: w ≡ u @: (v @: w)
```

is true when either side is applied to a noun.

To prove this, it is sufficient to notice that, when applied to a noun x, each side of the last Eq. works in J in the exactly same way: first is w applied to x producing a noun to which is v applied next producing a noun to which is u applied next producing a noun that is the result.

Figure1. Single object J category consisting of nouns (object N) and monadic verbs (indicated by subscript 1) as morphisms mapping N to N.

So far so good, and a single-element category within J is identified, consisting of an object *N* representing all nouns and having arrows that are all monadic verbs, with their composition given by `@:` and with identity arrow `]`, succinctly depicted in Figure 1.

All of this hopefully is rather simple and looks quite like standard composition of functions, just recast into the new terminology. There is, however, a subtle point at play: in J there are also verbs that are not functions at all: for instance the verb `?` (Deal), specifically designed to make up (pseudo-)random numbers, is a generator producing a different outcome every time the verb is invoked, and yet this does not affect **J** at all. This already gives us a tiny hint that category theory may be more general than just a theory of functions and their compositions.

3 Monadic verbs and their morphisms

Monadic verbs, just identified as morphisms of nouns, can be also viewed as an object *V1* of category **J**. Since most adverbs act on verbs to produce new verbs, they are natural candidates for morphisms mapping *V* to *V1*.

Composition of adverbs is provided by their concatenation, which produces a new adverb, and is also associative, i.e for every three J adverbs A, B and C that map monadic verb into monadic verb, the following holds

```
(A B) C ≡ A (B C)
```

due to the way J parser works [2], namely each side of the last Eq. acts on any verb so that A is applied first, B second and C third, regardless of parenthesis.

Identity morphism for verbs, id_{V1}, is an adverb idV1 satisfying the Eq. (1):

```
idV1 A ≡ A ≡ A idV1
```

for every adverb A mapping verb to verb. One solution is

```
idV1 =: ]@
```

and the proof as well as finding of other solutions are left as an exercise for the reader.

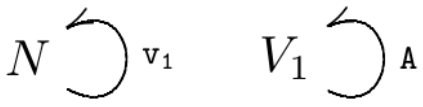


Figure 2. Two-object category of nouns and monadic verbs.

Thus, the second object of **J** has been found, that of all monadic verbs transforming by adverbs, which is graphically depicted in Figure 2. Such depiction, however, is missing a rather important structural information: although not incorrect, it does not represent the fact that adverbs are morphisms of verbs that are morphisms of nouns. The more precise view is given in Figure 3. It uses 2-category view of adverbs as morphisms of morphisms of nouns.

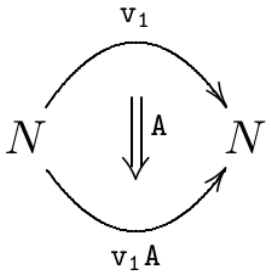


Figure 3. Categorical representation of J nouns, monadic verbs and adverbs.

4 What do dyadic verbs do?

In the previous section was shown that monadic verbs can be regarded as an object of a category, and that they are transformed by adverbs. In Sec. 2, monadic verbs were also seen as morphisms, mapping noun to noun. This mapping is after all their main function and use in J, and category theory here just formulates this by treating them as morphisms.

Suppose now that dyadic verbs are also morphisms of some object *X* of category **J**. In this case there exists identity arrow, id_X, represented in J as a dyadic verb idX such that for any dyadic verb v2 the following identity (corresponding to Eq. (1)) is true

```
idX @: v2 ≡ v2 ≡ v2 @: idX
```

The left identity is easily satisfied by taking `]` or `[` for `idX`, but the right identity has *no solution* in J. Therefore, either dyadic verbs are not morphisms of any object in J, or @: is not the correct composition of dyadic verbs, or both.

Another way to expose the same problem is if we understand dyadic verb as mapping a pair of nouns into noun, analogous to what monadic verbs do. Then we face a limitation of J insofar as there is no object in the language representing the pair.

It is possible to simulate a solution within J by packing two or in fact any number of nouns into a (boxed) array, and then unpack them when they are needed individually, which is the standard programming technique in J. The (boxed) array, however, no matter how many nouns it contains, is still a single noun, and therefore the verbs exchanging many nouns through boxed arrays, all the while capable to pass around very complex arguments, remain strictly monadic verbs.

Categorical view of J developed here allows to formulate the problem rather succinctly: category theory notion of morphism is an abstract representation of "doing something"; and while monadic verbs are morphisms, dyadic verbs are not.

4.1 Extension 1: Pairs

The plan is now to assume that @: is the operation of composition also of dyadic verbs and add a new class of J objects: the pair of nouns, with the corresponding object *NN* in **J** and provide as little as possible extra machinery that would make dyadic verbs become morphisms, mapping *NN* to *N*.

To this end is introduced a new verb `x.` (Pair) that takes two nouns and makes a pair out of them. J parser now must be explicitly extended with rules for handling *NN* pairs. A simple rule for handling the new object is parsing of V NN of the J stack that works exactly like the N V N parsing rule [2].

With the new object, verb and parsing rule, the following holds:

```
n v m ≡ v n x. m
```

for any dyadic verb `v`.

To see in details how the extension works, let us consider

```
1+2 ≡ + 1 x. 2
```

and follow in detail how parsing of the r.h.s unfolds.

First, `1 x. 2` is encountered on the J stack, which is parsed according to the standard N V N rule [2], producing a single NN object. Next, V NN is found on the stack, which is parsed by the new rule exactly as if the standard N V N (`1 + 2` in the example) were encountered, producing the result, number 3, on the stack.

Verb `x.` is also the identity arrow for *NN* , id_{NN}, because the following is true for any two nouns `n` and `m`

```
x: n x: m ≡ n x: m
```

This looks at first sight as if a roundabout way was taken, going from N V N through `x.` to V NN and then back again to N V N to achieve what was already done "directly" in J with V N V. Furthermore all of this machinery is seemingly introduced just so that trivially working identity morphism could be added to the language.

There is, however, a whole lot gained—for example, consider how the train `u x. v` works monadically and dyadically:

```
(u x. v) n ≡ (u n) x. (v n)

n (u x. v) m ≡ (n u m) x. (n v m)
```

Monadic `u x. v` therefore maps, via fork parsing rule V V V, noun to pair of nouns consisting of results of applications of the verbs to the noun argument "in parallel", and similarly for dyadic case. This could allow also something new to be done, namely to *define the fork* [6] via `x.` as

```
u v w ≡ v @: (u x. w)
```

which is nontrivial because it shows that once `u x. w` is introduced and parsed as above, the fork parsing rule could be made to be just a consequence of the way pair of verbs work.

Forks and verb pairs can also function together quite nicely with forks being evaluated serially as they currently are in J, while verb pairs evaluating two paired verbs in parallel, thereby providing an explicit mechanism in J for sorely lacking parallel programming.

Dyadic verbs have now become morphisms mapping *NN* to *N* in **J**, because it is easy to prove that earlier sought idV2 is now just the verb `x.` , with proof left as an exercise for the reader.

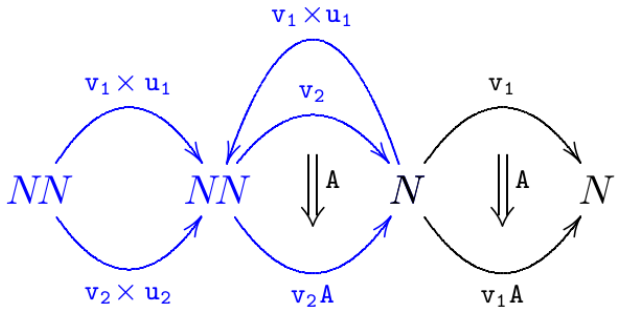


*Figure 4. Category **J** of J with the Extension 1. Additional structure induced by the Extension 1 is in blue.*

To summarize, the Extension 1 consists of: (i) adding a new class of J objects, noun pairs NN, constructed via verb `x:` ; (ii) adding the new parsing rule V NN. Both are manifestly orthogonal to the existing parsing rules of J.

Programming techniques that the extension provides, in addition to parallel evaluation mentioned earlier, are related to passing around pairs of nouns, as in the following verb that finds the minimal element of an array together with its position within the array:

```
min1 =: (i. x. ]) <./
```

where the verb returns a pair which first number is the position, the second the value of the minimum. This is superficially similar to the standard J solution that boxes the two numbers into an array:

```
min2 =: (i. ; ]) <./
```

with, however, important difference that, in the case of min1, no unboxing is necessary in further processing of results of min1 since the two nouns of the result are readily available as arguments to any dyadic verb.

5 What are conjunctions?

Next in order are conjunctions. They are similar to dyadic verbs insofar as they take two arguments and return one. While the dyadic verbs can be transformed using adverbs, it is not possible in J to do anything *with* conjunctions. They can be defined, but no language mechanism is provided for their transformation and/or composition. To include them into **J**, it is possible to make them objects or morphisms, or both, as is the case with dyadic verbs, for instance. In the former case there should be a mechanism for their transformation, while in the latter case, for their composition.

5.1 Extension 2: Conjunctions as objects

Here is explored the first case: conjunctions as object of **J**, in analogy to the dyadic verbs. Arrows of dyadic verbs are adverbs. The plan is now to make adverbs map conjunctions to conjunctions as well, through adding the C A bident rule.

An adverb that can be most straightforwardly extended to the new C A case is dyadic ~ (Reflex), that simply exchanges the left and right argument of the given verb, i.e

```
x v~ y ≡ y v x
```

with the obvious extension to the conjunction-adverb case simply given as

```
a C~ b ≡ b C a
```

for any conjunction C.

Given this, there is an identity morphism for conjunctions, id_C,

```
idC =: ~~
```

with a straightforward proof.

Here are two examples that use the Extension 2:

(i) The adverb ~ applied to conjunctions can be used for shortening of expressions as in:

```
f @ (u @ v) ≡ u @ v @~ f
```

which is similar in flavor to the commonly used

```
(a+b)%c ≡ c %~ a+b
```

(ii) Adverb @:~ inverts order in which two verbs are applied. Let us call it then then

```
then =: @:~
```

When connecting several verb with then, the overall effect is inversion of order in which they are applied:

```
f then g then h … then v ≡ v @: … h @: g @: f
```

where the l.h.s reads "Apply `f` then `g` … then `v`". The proof is straightforward and left as an exercise to the reader.

(iii) Changing of notation for `@.` (Agenda) according to the following identity

```
else`then@.cond ≡ cond@.~then`~else
```

In summary, the extension 2 consists of adding a parsing rule for C A bident, and extending at least ~ to work as described above with every conjunctions, implemented through the new parsing rule. Since the bident currently produces syntax error, its introduction will not change any working J program.

5.2 Extension 2': Conjunctions as morphisms

While the Extension 2 introduces conjunctions as an object of **J**, thereby providing a description of all J objects categorically in perhaps a minimal way, it still does not capture them as morphisms.

Before discussing how to achieve this in a general way, it is recalled that every conjunction with only one argument given becomes an adverb. This way each can be turned into an adverb and so automatically become a part of **J** built so far.

Here is an example using " (Rank) illustrating this role of conjunctions and the associated useful programming technique: verb `f` that calculates partial sums of row elements of an array:

```
f =: +/\"1
```

J parses the definition on r.h.s as `((+/)\)"1`., i.e first a verb is formed, and then the conjunction

applied using the formed verb and the number 1 [2]. It is, however, possible to abstract "doing partial something over rows" from `f` by defining the following adverb:

```
P =: /\("1)
```

which contains composition of three adverbs, / (Insert), \ (Scan) and "1, the last being an adverb that can be named Rank1 since it is based on conjunction " (Rank) "partially applied" to number 1.

P now allows that verbs for partial sums `f`, products `g`, and fractions `h`, of row elements of an arbitrary array be simply defined as

```
f =: + P
g =: * P
h =: % P
```

This way of looking at conjunctions as *parametric adverbs* comes quite naturally for many of them: apart from " (Rank), examples are: `L:` (Level At), `S:` (Spread), ;. (Cut), `D.` (Derivative), `d.` (Derivative), `D:` (Secant Slope), `T.` (Taylor Approximation), `H.` (Hypergeometric), `!.` (Fit), `!:` (Foreign) and `` `: `` (Evoke Gerund) [5]. Most uses of these can be interpreted using the identity

```
v C n ≡ v (C n)
```

which turns `C` into a member of the adverb family `C n`, just like in the example of Rank1 from a moment ago.

Another also large set of important conjunctions, however, do not fit into this scheme naturally because their working is perhaps best understood as *combining verbs* to produce another verb. Apart from @: (At), examples are: @ (Atop), `@.` (Agenda), `^:` (Power), `.` (Dot Product), .. (Even), `,:` (Odd), : (Explicit), &. (Under), &.: (Dual), and &: (Appose) [5]. Each of these is more suitably interpreted analogously to how dyadic verbs were formulated in the previous section: as mapping pair of verbs to verb.

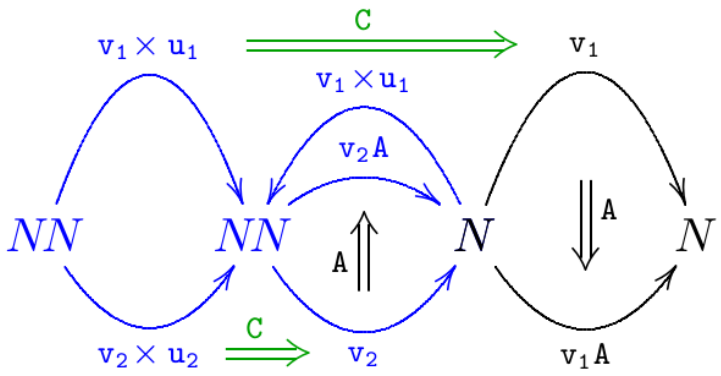


*Figure 5. Category **J** of J with the Extensions 1 (in blue) and part of the Extension 2' (in green.)*

Instead of going into details how to achieve this, only a sketch is provided here since the procedure is lengthy but similar to what was already done for dyadic verbs. The first step is recognizing all combinations of noun/verb pairs as new objects VV, VN and NV objects in J, (just like NN was introduced in the Extension 1.) This can be achieved by introducing `X.` conjunction. Then, additional

parsing rules are introduced to handle the new pairs in combination with conjunction, and orthogonally to the existing parsing rules. Perhaps the easiest way to achieve this is to introduce 4 new bidents: NN C, NV C, VN C and VV C into the J parser. These are manifestly orthogonal to both the current parsing rules of J [2] and the rules of Extensions 1 and 2.

The categorical structure will then be automatically inherited from the properties of adverbs as 2-morphisms (Sec. 3.) An interesting aspect of this approach is that conjunctions parsed this way can be interpreted as two-argument adverbs.

Figure 5 presents how **J** category looks like now, with Extensions 1 and 2', where only the conjunctions that combine two verbs are shown.

6 Conclusions and outlook

In this work a bare minimum of category theory was applied to analysis of J by representing its four classes of objects as elements of a category **J**. It is shown that this is possible only if two extensions are introduced. The first one is representation of dyadic verbs as mappings of pair of nouns into noun, which as a consequence also provides a definition of fork. The second extension is related to the inclusion of conjunctions into the categorical description, which was achieved in two ways: by treating them as objects of the category, or as mappings of pair of noun/verbs into noun/verb.

All three considered extensions are completely orthogonal both to the current J as well as among themselves, thus showing that J is quite conductive to this kind of language extensions. The route taken in this work is mainly oriented towards monoidal categories [7], which in practical terms means multi-argument verbs, adverbs and conjunctions, with emphasis on further development of J toward becoming a functional programming language.

7 References

[1] Dictionary of J: Terminology, http://jsoftware.com/help/primer/terminology.htm (accessed October 2013)

[2] Dictionary of J: E. Parsing and Execution, http://jsoftware.com/help/dictionary/dicte.htm (accessed October 2013)

[3] John C. Baez and Mike Stay, *Physics, Topology, Logic and Computation: A Roseta Stone*, http://math.ucr.edu/home/baez/rosetta/rose4.pdf, unpublished (2008)

[4] Michael Barr and Charles Wells, *Category Theory for Computing Science*, Prentice Hall Int. (1990)

[5] Dictionary of J: Appendix E. Parts of Speech, http://jsoftware.com/help/dictionary/partsofspeech.htm (accessed October 2013)

[6] Dictionary of J: Fork, http://jsoftware.com/help/primer/fork.htm (accessed October 2013)

[7] Peter Selinger, *A Survey of graphical languages for monoidal categories*, http://arxiv.org/abs/0908.3347, (2009)

J Functional Programming Extensions

Jose Mario Quintana and Thomas Costigliola

BEST, LLC, Miami FL, 33137 and Hoboken NJ, 07030 U.S.A.

More than a decade ago one of us decided to implement a large system tacitly. The standard J tacit programming features already available at the time made possible to generate tacit code of arbitrary verbs to perform any computable tasks since the tacit dialect of J is Turing complete [0, 1]. A major reason for favoring this approach is that it makes modular programming straightforward because fixed tacit code is stateless and point-free and consequently facilitates the production of non-interfering, self-contained, modules with a clear interface. However, it was apparent from the very beginning that some tacit extensions would make tacit programming easier and more powerful; after all, tacit J was not supposed to be abused to such an extreme. Recently we released some extensions that we have developed [2]; here, we review these J functional extensions and we introduce a few additional extensions that we intend to release in the near future.

1. The Primitive, Foreign and Train Extensions

We have allowed ourselves the liberty of defining some officially undefined primitives, instead of implementing them via foreign conjunctions, to produce more concise and readable programs. It might be advisable to use cover words for them in case these are officially defined differently in the future; in any case, that would be a momentous event.

Any unofficial extension is potentially controversial, to say the least, but some are quite more controversial than others; examples of the latter class are the extensions referred here as permissive. The hopefully less controversial extensions include the following words: the previously released recursion scope and 'word from linear' adverbs via the foreign conjunctions `(103!:0)` and `(104!:1)`; and three new primitives: a conjunction knot **(`.)** related to the tie (`) conjunction, and a trigger **(**`[.`**)** verb together with its complementary adverb trap **(**`].`**)** related to their explicit control counterparts break, continue and return. The permissive new words are the monadic form of the foreign apply verb **(**128:2) and the cloak (`]:`) primitive verb.

These words are shown in the following table (following the dictionary's style) and explained in the Appendix A0:

Primitive and Foreign Extensions

`(103!:0)`	***Recursion Scope*** ***mu lv rv***	`[.`	Trigger • _
`(104!:1)`	***Word from Linear***	`].`	***Trap*** ***mu lu ru***
`(128!:0)`	`Apply _ 1 _`	`]:`	Cloak (as verb) • Cloak _ _ _
`` `. ``	**Knot (Gerund)**		

The remaining extensions are syntactic in nature and include the (a v) form together with the resuscitated (a a), (c a) and (a c a) trains reverting to both, the meaning and behavior of J4.02d. These permissive trains are shown in the following table and explained in the Appendix A1:

Bidents and Tridents Extensions

a0 v1	***verb***	(*X*) ((*X*) v1 *Y*) ***a0*** *Y*
a0 a1	***adv***	(x ***a0***) ***a1***
c0 ***a1***	**conj**	(x **c0** y) ***a1***
a0 **c1** ***a2***	**conj**	(x ***a0***) **c1** (y ***a2***)

X and *Y* denote left and right (noun) arguments of a verb; (*X*) denotes optional (noun) left argument of an ambivalent verb; x and y denote left and right (noun or verb) argument(s) to a an adverb or a conjunction. Green coloring denotes permissive arguments .

A permissive argument can be any word (noun, verb, adverb or conjunction), as opposed to just a noun or a verb.

2. The Extensions in Action

(Stoop) if you are abecedminded, to this claybook, what curios of signs (please stoop), in this allaphbed! Can you rede (since We and Thou had it out already) its world? It is the same told of all. Many. Miscegenations on miscegenations. Tieckle. -- James Joyce, 'Finnegans Wake'

One could regard breaking the conventional rules of writing, by introducing alien words, blurring the differences between nouns, verbs adverbs and conjunctions, and mixing mixtures of parts of the speech, as an obfuscating and heretical form of expression; yet, someone else could find it disturbingly powerful and beautiful.

The primitive, foreign and train extensions are illustrated in the following J session with interjected commentary. The following sentences show how knot (`.) and the form (c a) facilitate tacit adverbial and conjunctional programming,

```
   u (w=. 'u^:v^:_' (conjunction :)          ) v NB. Explicit
u^:v^:_
   u (w=. `.((`.((<'^:')`._)) (@.0 2 1 2 3))) v NB. Tacit
u^:v^:_
   w
`. ((`.((<'^:'),<(,'0');_))(@.0 2 1 2 3))
```

Notice also the resistance of knot to the linear representation bug associated to tie (`),

```
   ` ((`((<'^:'),<(,'0');_))(@.0 2 1 2 3))
```

```
` ((`(^:`_))(@.0 2 1 2 3))
```

Let us continue by defining some preliminary utilities,

```
   (o=. @:) NB. Conjunction
@:
   (e=.&.>) (x=. @:[) (y=. @:]) (c=. "_) NB. Adverbs
(((&.>)(@:[))(@:]))("_)
   (lr=. 5!:5 o <) (ar=. 5!:1 o <) (dr=.5!:2 o <) (st=. 7!:2 y ;
6!:2) (sp=. 7!:5 o <) (j=. ,&<) NB. Verbs
5!:5@:< (5!:1@:< 5!:2@:< (7!:2@:] ; 6!:2) 7!:5@:< ,&<)
```

Next, we produce permissive dual verbalized versions of the adverb and conjunction primitives,

```
   ( (X=. 'power tilde dot even odd colon obverse adverse cut fit
foreign slash slashdot back backdot trap brace rank tie knot
evoke atop agenda at amper under dual appose') (;:x ,: ]) Y=. ;:
'^: ~ . .. .: : :. :: ;. !. !: / /. \ \. ]. } " ` `. `: @ @. @: &
&. &.: &:' )
[Suppressing  the output to save space.]

   ( (X)=. ]: o < e Y ) NB. Verbalizing primitive adverbs and
conjunctions
[Suppressing  the output to save space.]

   ( (X=. 'bdot derivative Derivative secant fix hypergeometric
LevelAt memo spread TaylorCoeff WeightedTaylor TaylorApprox')
(;:x ,: ]) Y=. ;: 'b. d. D. D: f. H. L: M. S: t. t: T.' )
[Suppressing  the output to save space.]

   ( (X)=. ]: o < e Y ) NB. Verbalizing the rest of the primitive
adverbs and conjunctions
[Suppressing  the output to save space.]
```

The following pair of verbs, one to evaluate trains (train) and another to atomize nouns (an), are quite useful,

```
   (train=. evoke&6) (an=. <@:((":0) j ]))
evoke&6 <@:((,'0') j ])
```

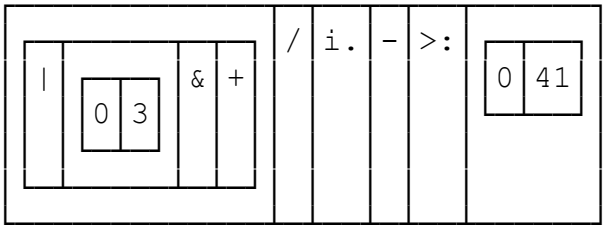
Consider an explicit verb that the tacit translator (13 :) refuses to process, for example, the version of Josephus verb in [3],

```
   3 (Josephus=. '(|x&+)/i.->:y' (4 :)) 41
30
   1000 st'3 Josephus 41'
```

3584	2.3556e_5

A corresponding tacit version can always be produced by means of sheer brute force, if necessary. First we build the atomic representation, so to speak, of the expression `(|x&+)/i.->:y` and use one of the hidden (undocumented) features of train (`` `:6 ``) afterwards,

```
   3 (ae=. (< o ((<'|') , (;:'&+') ,~ [) , (;:'/i.->:') , ])/ o
(,&:an) ) 41
```




```
   3 (Josephus=. train o ae f.) 41
30
   1000 st'3 Josephus 41'
```

3712	3.26170883e_5

```
   3 (Josephus=. apply o ae f.) 41
30
   1000 st'3 Josephus 41'
```

3712	3.20711787e_5

That is right, train and apply can evaluate atomic representations of expressions! (We had just implemented monadic apply when it became apparent that a verbalized version of (`` `:6 ``) was possible even using the official interpreter).

Let us consider yet another kind of representation, referred here as encase representation, of words which is both complementary and supplementary to the atomic representation, and an adverb utility (ew) to produce this representation,

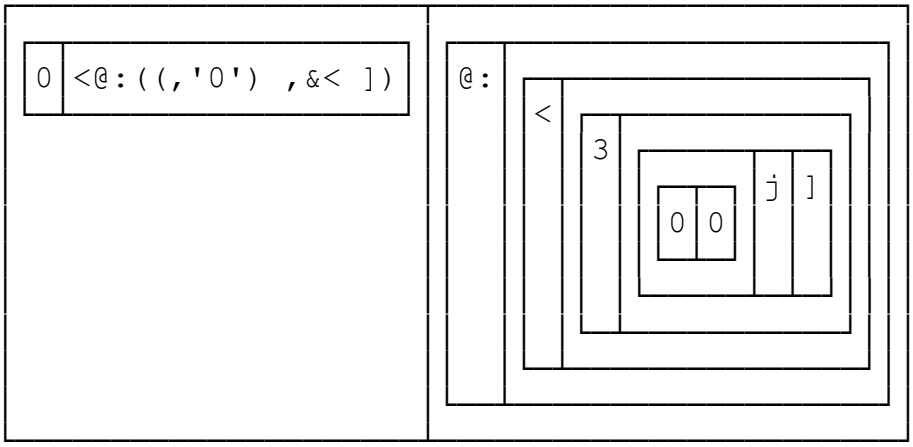
```
   ( ew=. adverb ]: (ar'an')  )
(1]:(<(<'@:'),<(<,'<'),<(<,'3'),<(<<;._1 ' 0 0'),(<,'j'),<,']'))
   ( ew=. adverb ]: an (f.ew) )
(1]:(<(,'0');<@:((,'0') ,&< ])))
```

The encase representation of a word is simply the atomized word, for example,

```
   (an o fix 'an') -: (an (f.ew))
1
```

The encase representation of a word is more compact and clearer than its corresponding atomic representation; compare, for example the linear representations of alternative versions of the verb `(ew)` above and the representations of the verb `(an)`,

```
   (an (f.ew)) , (ar'an')
```



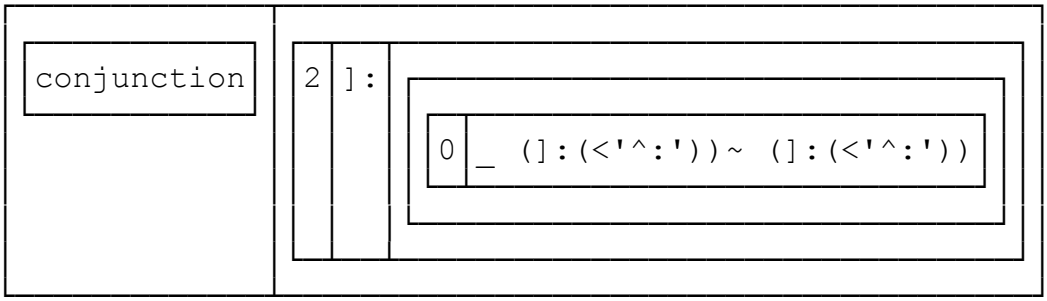
The encase representation of a word is also more efficient than its corresponding atomic representation because it is ready to go; more on this subject is presented later.

Let us produce a couple of adverbs that facilitate the (tacit) definition of adverbs and conjunctions,

```
   (adv=.  'ew'f. (adverb ]: adverb ((&]:) ew))) (conj=. 'ew'f.
(adverb ]: conjunction ((&]:) ew)))
((1]:(<(,'0');<@:((,'0') ,&<
]))))(1]:(<(,'0');1&]:)))((1]:(<(,'0');<@:((,'0') ,&<
]))))(1]:(<(,'0');2&]:)))
```

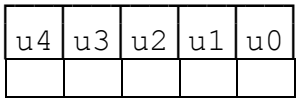
A simple example is to produce permissibly a concise alternative tacit version of the while conjunction `(w)` entertained earlier,

```
   u (w=. (_ power~ power)f. conj) v

u^:v^:_
   (type j dr) 'w'
```

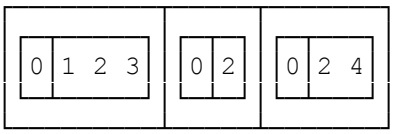


The following permissive sentences define an adverb to enable strand notation to produce trains of words; this adverb definition parallels its explicit counterpart in [5] that implements strand notation for nouns.

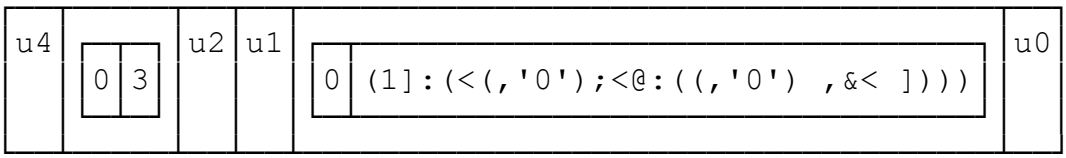
```
   cap=. (<'[:') (train x -: ]) [
   main=. (((<'main') train o , (an o knot)) ` (}:y)) @. cap conj
   sw=. main [:

   [: u4 u3 u2 u1 u0 sw
```




```
   [: '1 2 3' 2 (2 4) sw
```



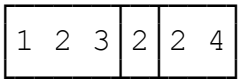

```
   [: u4 3 u2 u1 ('ew'(f.ew)) u0 sw
```




```
   train [: *: 1 2 3 sw
1 4 9
```

Strand notation for nouns, and composition of verbs, follow easily,

```
   (Ex=. < o train"0)  [: '1 2 3' 2 (2 4) sw
```




```
   (At=. (at~&:train)/ o |.) [: u4 u3 u2 u1 u0 sw
u4@:u3@:u2@:u1@:u0
```

Let us continue by illustrating uses of the less controversial trigger (`[.`) and trap (`].`) primitives; first, as an alternative to the control word return,

```
     i0=.  smoutput o ('Inner starts...'c)
     i1=.  [. o ([ smoutput) o ('Inner trigger value'c)
     i2=.  smoutput o ('Inner ends...'c)
   (inner=. (At[: i2 i1 i0 sw)].)
i2@:i1@:i0].

     o0=.  smoutput o ('Outer starts...'c)
     o1=.  inner
     o2=.  [ smoutput o ('Inner trigger return value:'c)
     o3=.  smoutput
     o4=.  smoutput o ('Outer continues...'c)
     o5=.  [. o ([ smoutput) o ('Outer trigger value'c)
     o6=.  smoutput o ('outer ends...'c)
```

```
   (outer=. (At[: o6 o5 o4 o3 o2 o1 o0 sw)].)
o6@:o5@:o4@:o3@:o2@:o1@:o0].

   outer _ NB. Here we go!
Outer starts...
Inner starts...
Inner trigger value
Inner trigger return value:
Inner trigger value
Outer continues...
Outer trigger value
Outer trigger value
```

second, as an alternative to the control word continue, following the Dictionary's example

(http://www.jsoftware.com/help/dictionary/ccont.htm),

```
   NB. Explicit...

   sumeven=: 3 : 0
 s=.0
 for_j. i.y do.
  if. 2|j do. continue. end.
  s=.j+s
 end.
)

   sumeven 9
20
   sumeven 1000
249500
   10 st'sumeven  1000'
```

7552	0.00422898

```
   NB. Tacit...

   ForBlock=. ( + [ [.y^:(2|[)  )].
   NB.      s=.j+s  if. 2|j do. continue. end.

   (sumeven=. i. ForBlock f. / o , 0:)
i. (+ [ [.@:]^:(2 | [))]./@:, 0:

   sumeven 9
20
   sumeven 1000
249500
```

```
   10 st'sumeven 1000'
```

9792	0.000625981

and third, as an alternative to the control word break, following the Dictionary's example (http://www.jsoftware.com/help/dictionary/cbreak.htm),

```
   NB. Explicit...

   itn=: 3 : 0 NB. Inverse triangular number
 s=.0
 for_j.
  i.2e4
 do.
  if. s>:y do. j break. end.
  s=.j+s
 end.
)

   Y=: 10 100 1000 10000
   itn "0 Y
5 15 46 142
   100 st 'itn  "0 Y'
```

134592	0.000726151

```
   NB. Tacit...

   'Y S J'=. 0 1 2

   ForBlock=. ((J&{ + S&{) S} ]) [ [. o (J&{)^:(S&{ >: Y&{)
     NB.        s=.j+s                        if. s>:y do. j
break. end.
   For_J=. ((1 + J&{ J} ])o) (^:2e4) (o(,0:))
   (itn=. ForBlock For_J f. o (,0:)].)
(1 + 2&{ 2} ])@:(((2&{ + 1&{) 1} ]) [ [.@:(2&{)^:(1&{ >:
0&{))^:20000@:(, 0:)@:(, 0:)].

   Y=: 10 100 1000 10000

   itn "0 Y
5 15 46 142
   100 st 'itn "0 Y'
```

1920	0.000468181

The sequence A008908 described in [6] can be defined and fixed using recursion scope `(103!:0)` as follows,

```
   rec=. 1:`(1 + $:@:(%&2`(1 + 3&*)@.(2&|)))@.(1&<)"0 M.
   (A008908=. rec o >: f.)
1:`(1 + $:@:(%&2`(1 + 3&*)@.(2&|)))@.(1&<)"0 M. (103!:0)@:>:
   A008908 o i. 71
1 2 8 3 6 9 17 4 20 7 15 10 10 18 18 5 13 21 21 8 8 16 16 11 24
11 112 19 19 19 107 6 27 14 14 22 22 22 35 9 110 9 30 17 17 17
105 12 25 25 25 12 12 113 113 20 33 20 33 20 20 108 108 7 28 28
28 15 15 15 103
```

Let us turn back our attention to the efficiency of the encase representation using the universal Turing machine verb `(utm)` in [1], as an example,

```
   ".@(('utm=. '),,)@(];._2)@(noun=. ".@('(0 : 0)'"_))_

((((":@:(]&:>)@:(6&({::)) ,: (":@] 9&({::))) ,. ':'"_) ,. 2&({::) >@:(((
48 + ]) { a."_)@[ ; (] $ ' '"_) , '^'"_) 3&({::))@:([ (0 0 $ 1!:2&2)@:(
'A changeless cycle was detected!'"_)^:(-.@:(_1"_ = 1&({::))))@:((((3&(
{::) + 8&({::)) ; 1 + 9&({::)) 3 9} ])@:(<@:((0: 0&({::)@]`(<@(1&({::))
@])`(2&({::)@])} ])@:(7 3 2&{)) 2} ])@:(<"0@:(6&({::) (<@[ { ]) 0&({::)
) 7 8 1} ])@:([ (0 0 $ 1!:2&2)@:((((":@:(]&:>)@:(6&({::)) ,: (":@] 9&({:
:))) ,. ':'"_) ,. 2&({::) >@:(((48 + ]) { a."_)@[ ; (] $ ' '"_) , '^'"_
) 3&({::))^:(0 = 4&({::) | 9&({::)))@:(<@:(1&({::) ; 3&({::) { 2&({::))
 6} ])@:(<@:(3&({::) + _1 = 3&({::)) 3} ])@:(<@:(((_1 = 3&({::)) {:: 5&
({::)) , 2&({::) , (3&({::) = #@:(2&({::))) {:: 5&({::)) 2} ])^:(-.@:(_
1"_ = 1&({::)))^:_)@:((0 ; (({. , ({: % 3:) , 3:)@:$ $ ,)@:(}."1)@:(".;
._2)@:(0&({::))) 9 0} ])@:(<@:('' ; 0"_) 5} ])@:(5&[] , a: $~ [))@:(,~)

)

   LR=. lr'utm' [ ER=. (utm (f.ew)) [ AR=. ar'utm'
   sp e o ;:'utm LR ER AR'
```

26688	1024	26880	53952

Regarding space, the linear representation is a small fraction versus the rest, the encase representation, not surprisingly, is very close to the internal representation and the atomic representation is the worst.

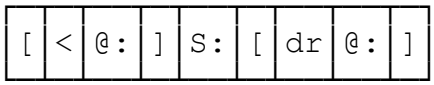
```
   wl=. 104!:1
   10 st e 'utm=. LR wl';'utm=. train ER';'utm=. train AR'
```

50880	0.000496451	1408	8.8944e_5	26496	0.000226613

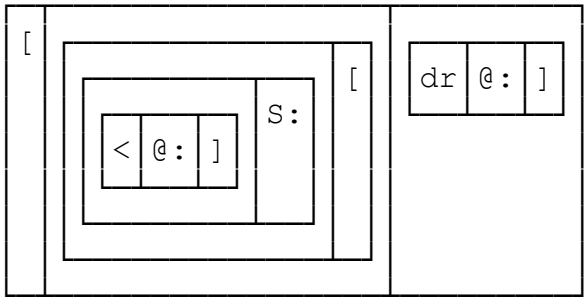
In terms of the space and time consumed when the verb is recovered, the atomic representation takes about half the time and space versus the linear representation, and the encase representation is the best by far.

The train `(a v)` provides an easy way to define conjunctions with calculated arguments, (for example, `X ((u c) v) Y` ↔ `X u c (X v Y) Y` as shown in [7]),

```
   test=. [ ((<y S:)[) dr y
    0 test 'test'
```




```
   _1 test 'test'
```



Finally, the following sentences are a tacit implementation of a slight variation of the quantum "commute" operator `(comm)` in [8, 9]; they show an application of the train `(a c a),`

```
   ddx=. d.1            NB. - first derivative (adv)
   xmul=. (]`*`) (`:6) NB. - multiply by x    (adv)

   minus=. train o ([ tie -`'' tie ]) conj
   comm=. ((< o (j~) , ('minus' (f.ew)) , < o j)conj) (train adv)

           'ddx' comm 'xmul'
(xmul ddx) (2]:(<(,'0');train@:([ tie (<'-') tie ]))) (ddx xmul)

         % 'ddx' comm 'xmul' %
(% + ] * -@%@*:) - ] * -@%@*:

         % 'ddx' comm 'xmul' %  i.5
_ 1 0.5 0.333333 0.25

   (% -: % 'ddx' comm 'xmul' %) i.5
1
```

Appendices

A0. Primitive and Foreign Extensions

rs Recursion Scope (103!:0)

Is an adverb; `u 103!:0` is the same as u except that `$:` in u refers to u and not to the entire verb phrase that contains `u 103!:0`. Moreover, applying `f.` to a verb phrase that contains u

`103!:0` does not change its result. Finally, `v f.` will be suffixed with `103!:0` rather than "wrapped" in an explicit definition, if `v` contain `$:`.

wl Word from Linear (104!:1)

Is an adverb; `x wl` executes string `x` and returns the noun, verb, adverb, or conjunction result, or the empty string if there is no result.

apply Monadic Apply (128!:0)

The monadic form of the verb apply executes atomic (or encase) representations of words and expressions.

`. Knot

Is a conjunction; `` `. `` is similar to the J primitive tie (`) except that a non-boxed noun argument is first replaced by its atomic representation. This is the suggestion in [4] except for the treatment of non-boxed nouns.

`[.` and `].` Trigger and Trap

`].` (Trap) is an adverb. `u ].` Is the same as `u`, except that if the verb `[.` (Trigger) is encountered, `u` returns immediately with the result being the monadic argument to the `[.` encountered.

]: Cloak

`]:` is a verb. `X ]: Y` returns the functional whose atomic representation is Y as a verb (3), adverb (1) or conjunction (2) depending on the return code `X. ]: Y` is equivalent to `3 ]: Y` and it is also equivalent to `(<(<,'4'),<(<(<,'4'),<;:'0:`'),<(<,'4'),<;:',^:')(0:`),(,^:)` as explained in [10] and the references there in .

A1. Train Extensions

There is only one extra row, the trident, added to the standard Parse Table but some extra combinations, shown in the Bidents and Tridents Extensions Table, are recognized by the interpreter. This modified Parse Table follows:

Parse Table

EDGE	verb	*NOUN*	any	0 Monad
EDGE/AVN	verb	verb	*NOUN*	1 Monad
EDGE/AVN	*NOUN*	verb	*NOUN*	2 Dyad
EDGE/AVN	verb / *NOUN*	***adv***	any	3 Adverb
EDGE/AVN	verb / *NOUN*	**conj**	verb / *NOUN*	4 Conjunction
EDGE/AVN	verb / *NOUN*	verb	verb	5 Fork
EDGE	**ca***VN*	**ca***VN*	**ca***VN*	6 Trident
EDGE	**ca***VN*	**ca***VN*	any	7 Bident
NOUN-NAME	**=. / =:**	**ca***VN*	any	8 Is
(	**ca***VN*	**)**	any	9 Paren

Legend: **av*N*** denotes **Adv** / verb / *NOUN*
ca*V*N denotes **conj** / ***adv*** / verb / *NOUN*
EDGE denotes ***MARK*** / **=. / =: / (**

J functional extension

Bident `(a v)`

This train, proposed in [7], is implemented as `(X )(a v)Y ↔ (X )((X ) v1 Y )a0 Y`

Bident `(a a)`

This train restores the implementation `x (a0 a1) ↔ (x a0 ) a1.`

Bident `(c a)`

This train restores the decommissioned implementation, `x (c0 a1) v ↔ (x c0 y) a1`

Trident `(a c a)`

This train restores the decommissioned implementation, `x (a0 c1 a2) y ↔ (x a0 ) c1 (y a2 ).`

References

[0] A Tacit Implementation of a Turing Machine, Jforum,
http://www.jsoftware.com/pipermail/general/1999-December/002736.html

[1] Universal Turing machine, J implementation of the Rosetta Code task,
http://rosettacode.org/wiki/Universal_Turing_machine#J

[2] J functional programming extensions, Jprogramming forum,
http://www.jsoftware.com/pipermail/programming/2013-March/031835.html

[3] Rosettacode, Jprogramming forum,
http://www.jsoftware.com/pipermail/programming/2013-May/032435.html

[4] Tie question (suggestion?), Jprogramming forum,
http://www.jsoftware.com/pipermail/programming/2006-October/003497.html

[5] Third argument (was: avoid boxing with fills - ?), Jprogramming forum,
http://www.jsoftware.com/pipermail/programming/2009-July/015541.html

[6] Number of halving and tripling steps to reach 1 in the Collatz (3x+1), OEIS A008908,
http://oeis.org/A008908

[7] Atop u at v with v of negative monadic rank, Jprogramming forum,
http://www.jsoftware.com/pipermail/programming/2010-November/021108.html

[8] Conjunction of adverbs, Jprogramming forum,
http://www.jsoftware.com/pipermail/general/2003-December/013430.html

[9] Conjunction of adverbs, Jprogramming forum,
http://www.jsoftware.com/pipermail/general/2003-December/014533.html

[10] J functional programming extensions, Jprogramming forum,
http://www.jsoftware.com/pipermail/programming/2013-March/031883.html

Solving Maximum Flow Problems in J

Michal Dobrogost *

1 Abstract

After a brief introduction to maximum flow problems, a solution is formulated in J. The existing family of push-relabel algorithms is refined in two ways to better suit the high-level nature of J. The focus is on correctness and pedagogy not outright execution efficiency. We conclude by tracing the execution over a specific maximum flow instance. A proficiency in the J programming language is assumed.

2 Introduction

Maximum flow problems were first formulated in 1954 by T.E. Harris to model Soviet railway traffic. The motivation was to find the maximum carrying capacity of the railway system between two cities. An additional motivation was the dual, minimum cut, problem which would allow one to determine how to disconnect a railway network most efficiently [1].

Solving maxflow has applications in bipartite matching, vertex cover, scheduling, baseball elimination, image segmentation and many others [6, 7, 4]. I was lead down the path of implementing a solver in J while exploring network robustness measures and the edge connectivity of a graph in particular. Network robustness is concerned with qualifying how reliable a given network is in the face of failing edges or nodes. Edge connectivity is one such measure and is defined as the minimum number of edges that need to be removed from a graph in order to disconnect it [5].

Maximum flow problems can be defined as follows: given a graph, two distinguished nodes, s and t, and a capacity constraint on each edge find the maximum amount of flow from s to t through the edges of the graph so that the flow through any edge does not exceed its capacity.

*michal dot dobrogost at gmail
www.cucave.net/papers/jmaxflow

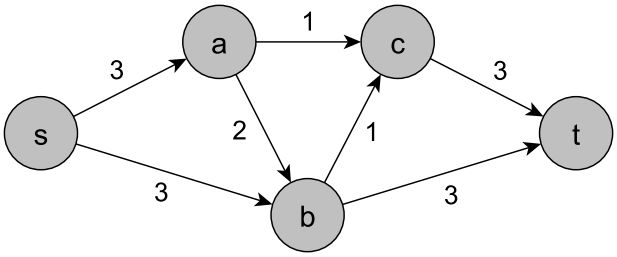


Figure 1: A maximum flow instance with edges labeled by their capacity. This graph will serve as a running example.

To be more precise, given a graph $G = (V, E)$, nodes $s, t \in V$ and a function $C \colon V \times V \to \mathbb{R}+$ find a function $F \colon V \times V \to \mathbb{R}$ such that:

$$\forall x, y \in V, F(x, y) \leq C(x, y) \qquad \text{(honour edge capacities)}$$

$$\forall x \in V \setminus \{s, t\}, \sum_{z \in V} F(z, x) - F(x, z) = 0 \qquad \text{(nodes conserve flow)}$$

$$\forall x, y \in V, F(x, y) = -F(y, x) \qquad \text{(skew symmetry)}$$

$$\arg\max_F |F| = \sum_{x \in V} F(s, x) = \sum_{x \in V} F(x, t) \qquad \text{(maximize flow amount)}$$

The instance in Figure 1 is essentially encapsulated in a single $|V| \times |V|$ matrix where the entry at C_{ij} represents the maximum amount of flow allowed along the edge from node i to node j.

```
   C;s;t
+---------+-+-+
|0 3 3 0 0|0|4|
|0 0 2 1 0| | |
|0 0 0 1 3| | |
|0 0 0 0 3| | |
|0 0 0 0 0| | |
+---------+-+-+
```

The first three properties and the summation of the fourth can be readily expressed in J. The maximization of the fourth, while satisfying the others, is the goal of this paper. The flow F is expressed as a matrix of the same shape as C.

```
edgeCapSatisified=.  *./ , F <: C
conserveFlowSatisified=. *./ 0= (-. (i.#F) e. s,t) # (+/ - +/"1) F
skewSymSatisified=. *./ 0= , F + |:F
flowAmount=. +/ {. F
```

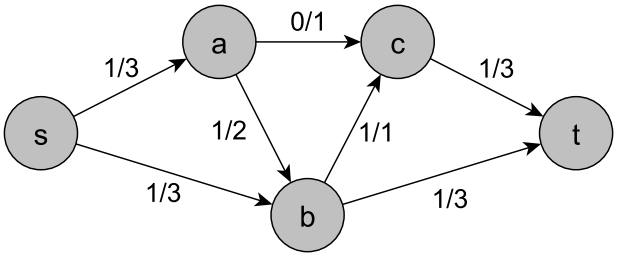


Figure 2: A flow is shown. Edges are labeled with their *flow/capacity.*

3 The Residual Graph

An implicit choice was made in that the domains of C and F were defined as $V \times V$ and not E. Suppose you have two nodes that have a single edge between them (for example a and b). Once flow is sent along that edge, from a to b, you can now send flow in the other direction, from b to a, by decreasing the amount of flow on the original edge. Essentially, any flow F will induce a residual graph in such a way. This is demonstrated in Figures 2 and 3.

```
   ] F=. 5 5 $ 0 1 1 0 0, 0 0 1 0 0, 0 0 0 1 1, 0 0 0 0 1, 0 0 0 0 0
0 1 1 0 0
0 0 1 0 0
0 0 0 1 1
0 0 0 0 1
0 0 0 0 0
```

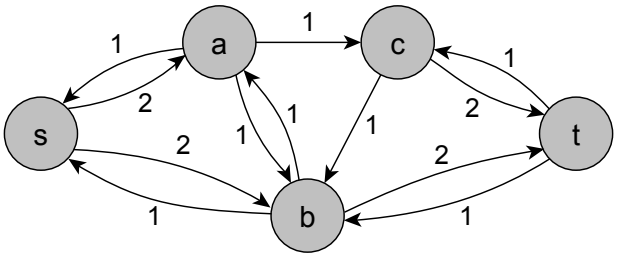


Figure 3: The residual graph induced by the flow in Figure 2. The edges are labeled with their residual capacity. This is the only time that the residual graph is made explicit.

```
   ] F =. (skewSym=. - |:) F
 0  1  1  0 0
_1  0  1  0 0
_1 _1  0  1 1
 0  0 _1  0 1
 0  0 _1 _1 0
   ] Residual=. C - F
0 2 2 0 0
1 0 1 1 0
1 1 0 0 2
0 0 1 0 2
0 0 1 1 0
```

During the execution of the algorithm it is useful to be able to update only overflowing nodes. A skew symmetric update is required for which we introduce the `updateSkewSym` function. In this example we push an additional unit of flow from nodes s and a, to b.

```
   updateSkewSym=. 4 : 0
    'f'=. x
    'vs fDelta'=. y
    f + skewSym fDelta vs} ($f) $ 0
)
   F ; F updateSkewSym (0,1);(2 5 $ 0 0 1 0 0)
+-------------+-------------+
| 0  1  1  0 0| 0  1  2  0 0|
|_1  0  1  0 0|_1  0  2  0 0|
|_1 _1  0  1 1|_2 _2  0  1 1|
| 0  0 _1  0 1| 0  0 _1  0 1|
| 0  0 _1 _1 0| 0  0 _1 _1 0|
+-------------+-------------+
```

4 Algorithm Prerequisites

There are many possible approaches to solving maxflow (an overview is presented in [2]). In practice push-relabel algorithms are the fastest [3] and an instance of this family is presented here.

Push relabel algorithms relax the flow conservation property by maintaining a preflow instead. A preflow has the requirement that, for internal nodes, more flow may come in than leave and if this is the case the node is referred to as overflowing.

$$\forall x \in V \setminus \{s,t\}, \sum_{z \in V} F(z,x) - F(x,z) \geq 0 \qquad \text{(preflow)}$$

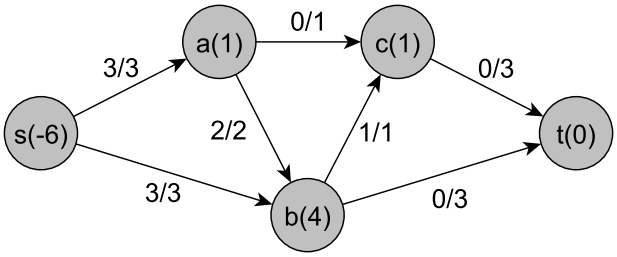


Figure 4: A preflow is shown. Edges are labeled with *flow/capacity*. Nodes are labeled with *name(excess)*.

The excess flow coming into a node may be computed using a difference of the flow matrix to its transpose. The definition of `vertexExcess` takes a skew symmetric flow as input and uses the rank changing operator `"1` instead of transpose.

```
   ] F=. 5 5 $ 0 3 3 0 0, 0 0 2 0 0, 0 0 0 1 0, 0 0 0 0 0, 0 0 0 0 0
0 3 3 0 0
0 0 2 0 0
0 0 0 1 0
0 0 0 0 0
0 0 0 0 0
   (vertexExcess=. 0.5 * (+/) - (+/"1)) skewSym F
_6 1 4 1 0
```

A height function $H\colon V \to \mathbb{N}_0$ labels the nodes and restricts where an overflowing node's flow may go next. The intuition is that flow may only go downhill from a node of higher height to a node of lower height.

$$\begin{aligned} H(s) &= |V| \\ H(t) &= 0 \\ (C-F)(u,v) > 0 \implies H(u) &\leq H(v)+1 \end{aligned}$$

A utility function is used for restricting a vector up to a given cumulative sum:

```
   untilAddTo=. 4 : 0
    (<.&y) &.: (+/\"1) x
)
   (i.2 7) ([ ; untilAddTo ; (+/"1 @: untilAddTo)) 7,21
+-----------------+-------------+----+
|0 1 2  3  4  5  6|0 1 2 3 1 0 0|7 21|
|7 8 9 10 11 12 13|7 8 6 0 0 0 0|    |
+-----------------+-------------+----+
```

5 Algorithm

The algorithm is initialized by pushing as much flow as possible from s to all adjacent nodes. All heights are initialized to zero except for $H(s) = |V|$. The return type here serves as the input and return type for the other verbs in this section, this can be seen as the current state of the execution.

```
   NB. Initialize push-relabel data structures
   NB. Returns (flow, residual capacities, vertex heights, vertex excess)
   init=. 3 : 0
    'C s t'=. y   NB. (original capacities, index of s, index of t)
    n=. #c

    NB. Initialize flow by sending out as much as possible from s.
    F=. (($C)$0) updateSkewSym s ; s{C

    NB. Initalize heights by setting s height to n, all others to 0.
    H=. (n,0) (s,t)} n $ 0

    NB. Put them all together
    F;(C - F);H;(vertexExcess F)
)
```

The remainder of the algorithm is a repetition of two operations: `relabel` and `push`. In `relabel` overflowing nodes are found whose height is insufficient to push out excess flow. The height is then increased to the maximum allowed by the definition of the height function which is governed by the residual neighbors of a node.

```
   relabel=. 4 : 0
    'C s t'=. x
    NB. (original capacities, index of s, index of t)
    'F R H E'=. y   NB. (flow, residual, vertex heights, vertex excess)

    NB. Active (overflowing) nodes
    a=. 0 0 (s,t)} E > 0
    aIx=. a # i. #c

    NB. Update height of active nodes by one more than their min neighbor
    H2=. (1+ (0 < a#R) (<./ @ #)"1 H) aIx} H

    F;R;H2;E
)
```

In `push` as much excess flow as possible is pushed out of an overflowing node. Only neighboring nodes to which there is a residual edge and whose height is one less than the overflowing node's

height may accept the excess flow. Because of the constraints on the height function, a height difference of one is the same as pushing excess flow to neighboring nodes of lower height. We use `untilAddTo` to push the excess flow to as many neighbors as possible - limited by the excess amount overflowing or by the residual capacity of the outgoing edges.

```
   push=. 4 : 0
    'C s t'=. x     NB. (original capacities, index of s, index of t)
    'F R H E'=. y   NB. (flow, residual, vertex heights, vertex excess)

    NB. Active (overflowing) nodes
    a=. 0 0 (s,t)} E > 0
    aIx=. a # i. #c

    NB. Selection of active vertex edges which have positive residual capacity
    edgesPosCapacity=. 0 < a#R

    NB. Selection of active vertex edges to nodes of height smaller by one
    edgesHDiff=. 1 = (a#H) (-"0 _) H

    NB. Residual capacity of active vertex edges which can take more flow
    edges=. (a#R) * edgesPosCapacity * edgesHDiff

    NB. Add flow along the first residual edges but limited by the excess amount
    flowToAdd=. edges untilAddTo (a#E)

    NB. Add the flow
    F2=. F updateSkewSym aIx;flowToAdd

    NB. Find qualifying edges for each active vertex
    F2;(C - F2);H;(vertexExcess F2)
)
```

Finally we provide a wrapper that initializes the data structures and then alternatively runs `relabel` and `push` operations until the data structures stop changing. Once that happens, progress has stopped and the algorithm is complete.

```
   maxflow=: 4 : 0
    'C'=. x
    's t'=. y
    'F R H E'=. ((C;s;t)&push @: ((C;s;t)&relabel))^:_ init (C;s;t)
    F
)
```

```
   c maxflow s;t
 0  2  3  0 0
_2  0  1  1 0
_3 _1  0  1 3
 0 _1 _1  0 2
 0  0 _3 _2 0
```

6 Correctness

The algorithm was tested against *solver-5* from the *First DIMACS Implementation Challenge*[1]. In all tested cases the resulting flow amount matched that of the other solver. A DIMACS format reader in J has also been implemented [2].

Full correctness proofs for the push-relabel family of algorithms can be found in [3]. Here we present the atomic description of `push` and `relabel` as given in the full proofs. The atomic operations apply to a single node or edge and can be applied so long as preconditions are met. Correctness is proved here by showing that extending these operations to work over all possible inputs they apply to is also correct.

`relabel(u)`
Precondition: u is overflowing, $(C - F)(u, v) > 0 \implies H(u) \leq H(v)$.
Effect: $H(u)$ is increased to one more than the minimum $H(v)$ where $(C - F)(u, v) > 0$.

`push(u,v)`
Precondition: u is overflowing, $(C - F)(u, v) > 0 \implies H(u) = H(v) + 1$.
Effect: push the minimum of $E(u)$ or $(C - F)(u, v)$ units of flow from u to v.

Relabeling multiple nodes

It is correct to apply `relabel` over all nodes to which it applies.

The major difference is that we do not check the second precondition, however, this has no effect except for additional runtime costs. By the definition of H we have that $(C - F)(u, v) > 0 \implies H(u) \leq H(v) + 1$. Since the height is updated to one more than the minimum we know that the height will not change at all (if $H(u) = H(v) - 1$) or will increase otherwise. So the operation may be applied and produce no change, but that is the same as not applying it at all because the precondition is not met. This is the same progress condition as the atomic `relabel` in [3] provides.

Since the atomic operation may be applied in any order, and only a single node's entry in H is updated by the atomic version, it is safe to apply the operation to all overflowing nodes at once and is equivalent to any ordering of sequential `relabel` operations.

[1] `http://dimacs.rutgers.edu/Challenges/`
[2] `www.cucave.net/papers/jmaxflow`

Pushing excess out of multiple nodes

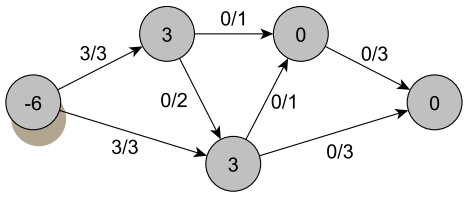
It is correct to apply `push` over all nodes to which it applies.

The preconditions are checked as listed for the scalar case. Pushing can remove and add edges in the residual graph as well as modify their capacities. The task is to ensure that pushing out of multiple nodes will not cause conflicts.

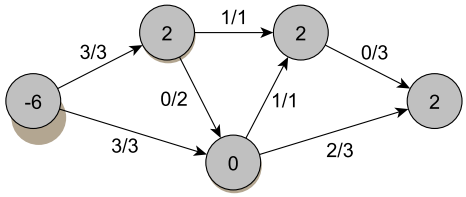
The only type of edge that is removed has its starting point in the overflowing node. The only type of edge that is added has its end point in the overflowing node. Both types of edges can be modified. Since a height difference of one is required only one node out of any pair of nodes connected in the residual graph will satisfy the preconditions to push flow to the receiving node. This is enough to ensure that each scalar operation affects a different set of edges than any other scalar operation that applies at the same time.

Pushing does not change the height of a node and so it does not affect its own preconditions that way. However, decreasing the excess or making a choice between multiple different targets of the push may affect what pushes are possible in the future. This is an arbitrary choice and so there exists a set of atomic push operations which will have the same effect as applying them all at once so long as the same choices of source and target nodes is made.

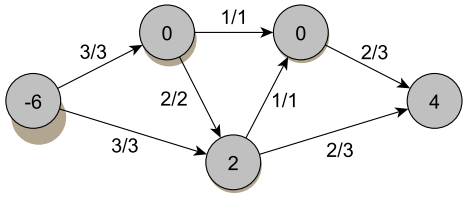
7 Example



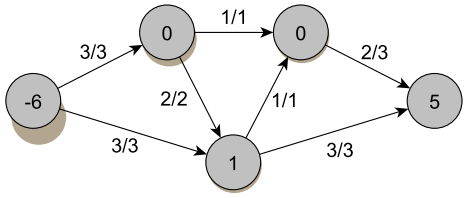
(a) State after initialization.



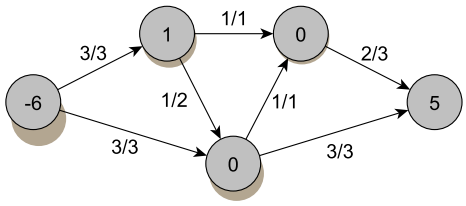
(b) Node B splits its excess onto both of its neighbors. This is because node c comes before t in the node ordering.



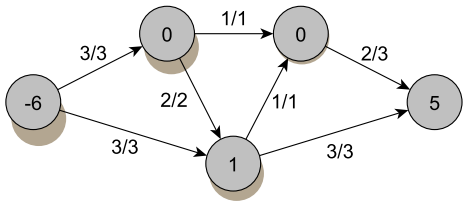
(c) Node A can push to B now that there is a height differential.



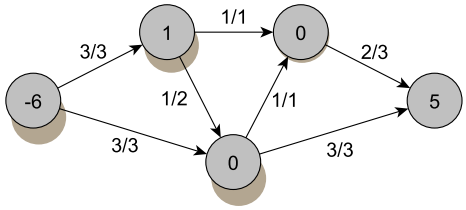
(d) Node B pushes remaining flow to T as limited by the edge capacity.



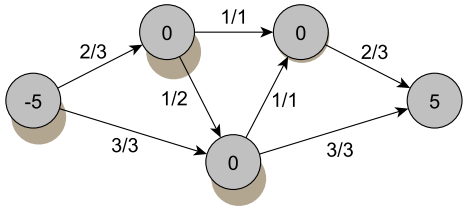
(e) Nodes A and B now swap excess until they achieve sufficient height.



(f) Nodes A and B swapping.



(g) Nodes A and B swapping.



(h) Node A has achieved sufficient height to push back to S.

Figure 1: Tracing a complete run. Edges are labeled with *flow/capacity* while nodes are labeled with *excess*. Vertex height is indicated by the length of a node's shadow. Each step consists of one relabel and one push operation.

8 Conclusion

An extension of push-relabel algorithms was made to apply the atomic operations to multiple nodes at once. This is a requirement for efficient implementation in J. An arbitrary ordering of operations results in a $\mathrm{O}\left(|V|^2|E|\right)$ time bound for push-relabel algorithms. Better bounds exist when the order of scalar operations is carefully controlled. This implementation is worse than the baseline bound because some individual steps were implemented for clarity:

- `updateSkewSym` builds an entire flow matrix instead of updating individual rows of the current matrix.
- `untilAddTo` has to run over a representation where all nodes are listed and not just possible residual neighbors. A more sparse graph representation would need to be chosen.
- `vertexExcess` recomputes the excess amount by summing all edges of each node, a more incremental approach would be more efficient.

Writing a solver directly in J has illuminated the details of push-relabel algorithms. It has also shaped the initial thinking - elucidating the idea of preflow before I dove into existing literature. Framing the algorithm in J naturally resulted in identifying potential areas of parallelism.

References

[1] Alexander Schrijver *On the history of the transportation and maximum flow problems.* Mathematical Programming, February 2002, Volume 91 Issue 3, pages 437-445.

[2] Andrew V. Goldberg, Robert E. Tarjan *A new approach to the maximum-flow problem.* Journal of the ACM Oct. 1988, Volume 35 Issue 4, pages 921-940.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein *Introduction to Algorithms (*2^{nd} *edition)* MIT Press and McGraw-Hill 2001.

[4] Ravindra K. Ahuja, Thomas L. Magnanti, James B. Orlin *Network Flows: Theory, Algorithms, and Applications* Prentice-Hall, Inc. Upper Saddle River, NJ, USA 1993.

[5] Wendy Ellens *Effective resistance and other graph measures for network robustness* PhD Thesis, Mathematisch Instituut at the University of Leiden 2011.

[6] Jeff Erickson *Algorithms Lecture 23 : Applications of Maximum Flow* `http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/23-maxflowapps.pdf` Fall 2010.

[7] Nikhil Bansal *Maximum Flow Applications* `http://www.win.tue.nl/~nikhil/courses/2WO08/max-flow-applications-4up.pdf` Winter 2008.

Journal of J

An interdisciplinary journal on J programming language and applications in science and technology.

www.journalofj.com

THE J GUIDEBOOK for Programming, Numerics and Graphics.

A international and interdisciplinary challenge !

This is a new and stimulating project of JoJ team and J community. The challenge is to create a companion (or book, or algorithm repository) to the work *The art of computer programming*. Similarly, in this project, we intend to create a work similar to *The Mathematica Guidebook* of Michael Trott. In this context ,the work of Michael is a source of ideas to create verbs, programs and scripts in J.

For more information send a e-mail to info@journalofj.com.

****

****

****

****

*

MPM Press **AN OPEN JOURNAL**

ISSN: 2174-9280