Loops solving in OpenRC

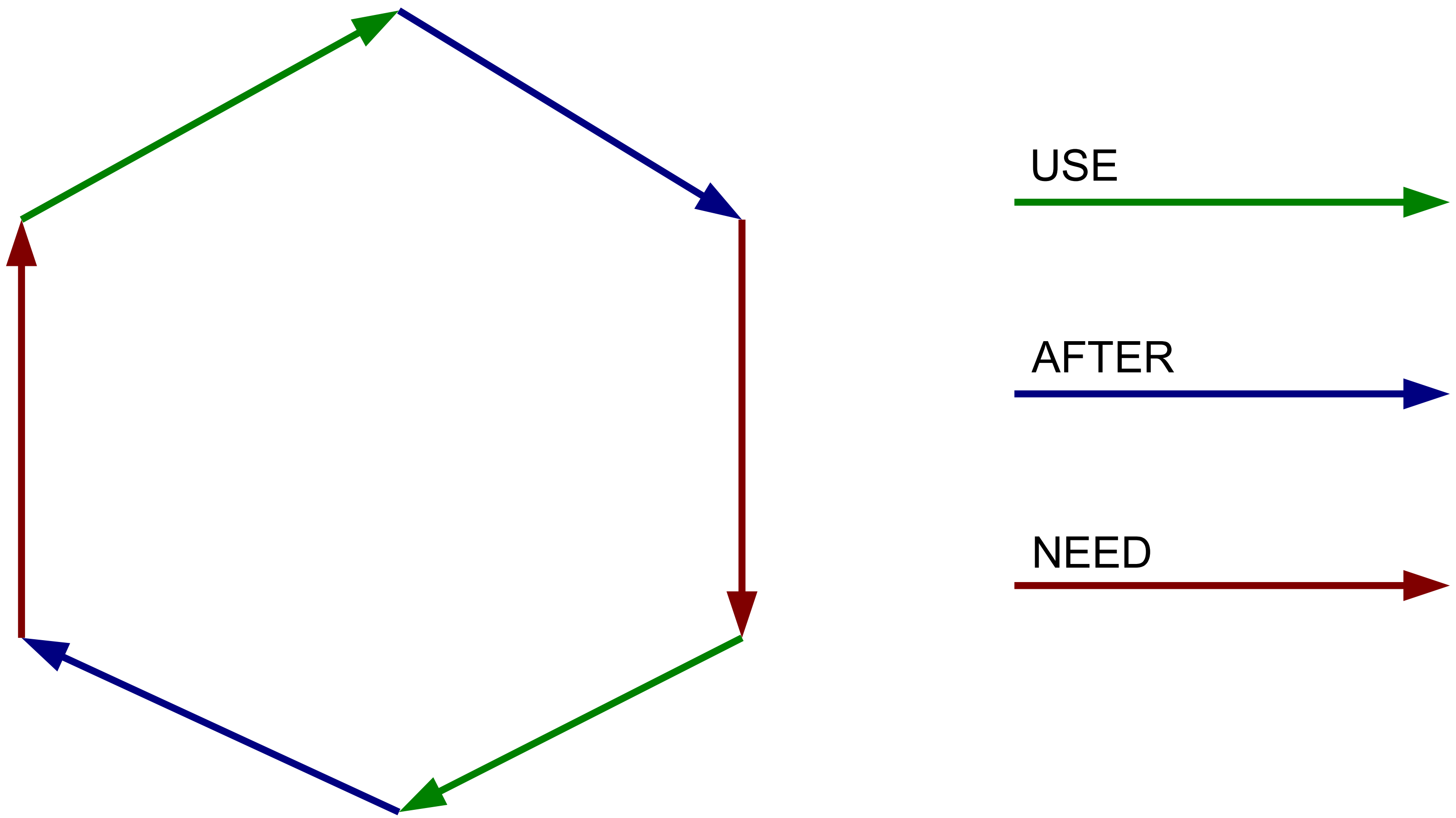
Dmitry Yu Okunev <dyokunev@ut.mephi.ru> 0x8E30679C

Explanation of methods that I used to detect and solve loops in my patch for OpenRC “early loop detector”.

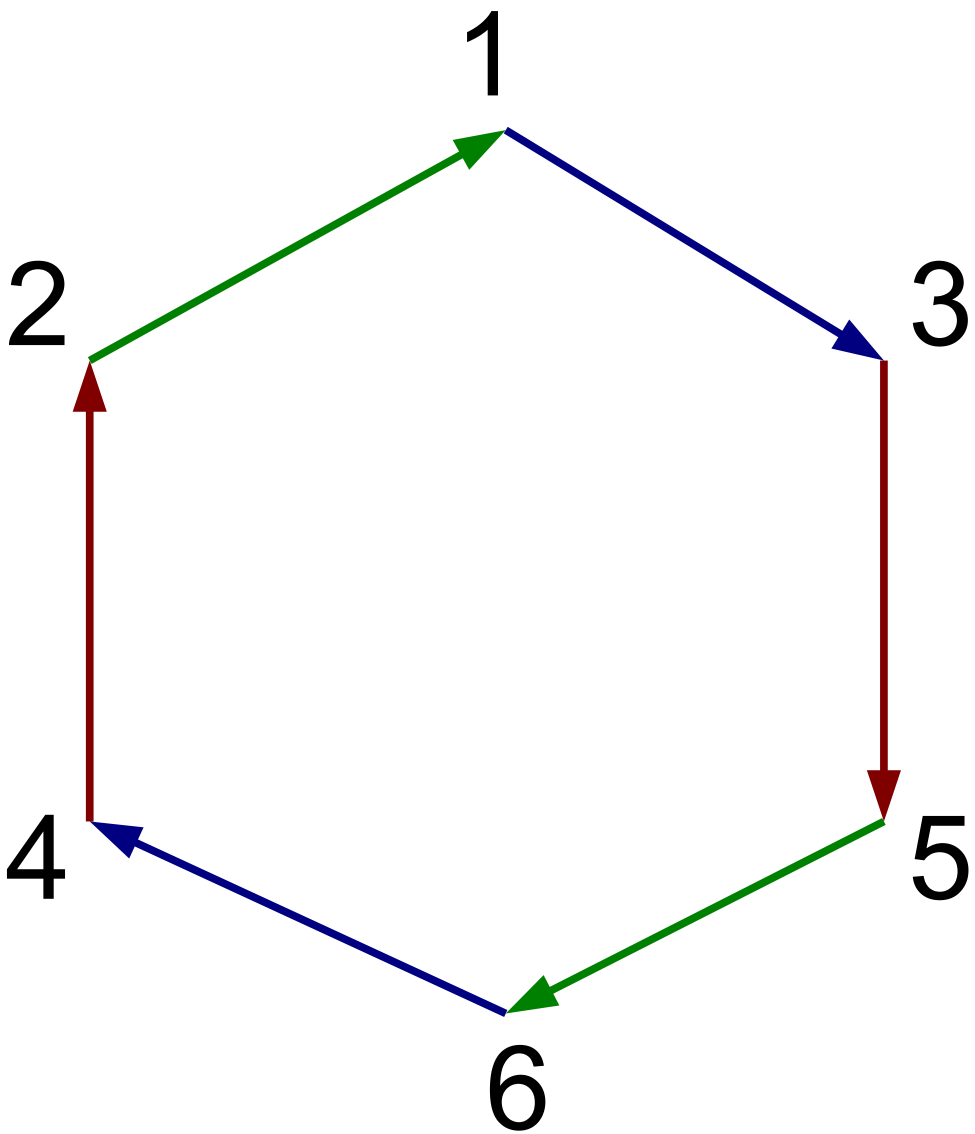
I don't know how are this algorithms called, because I solved the problems on piece of paper by myself. Sorry…

Also please sorry for my English.

Let's imagine a simple loop

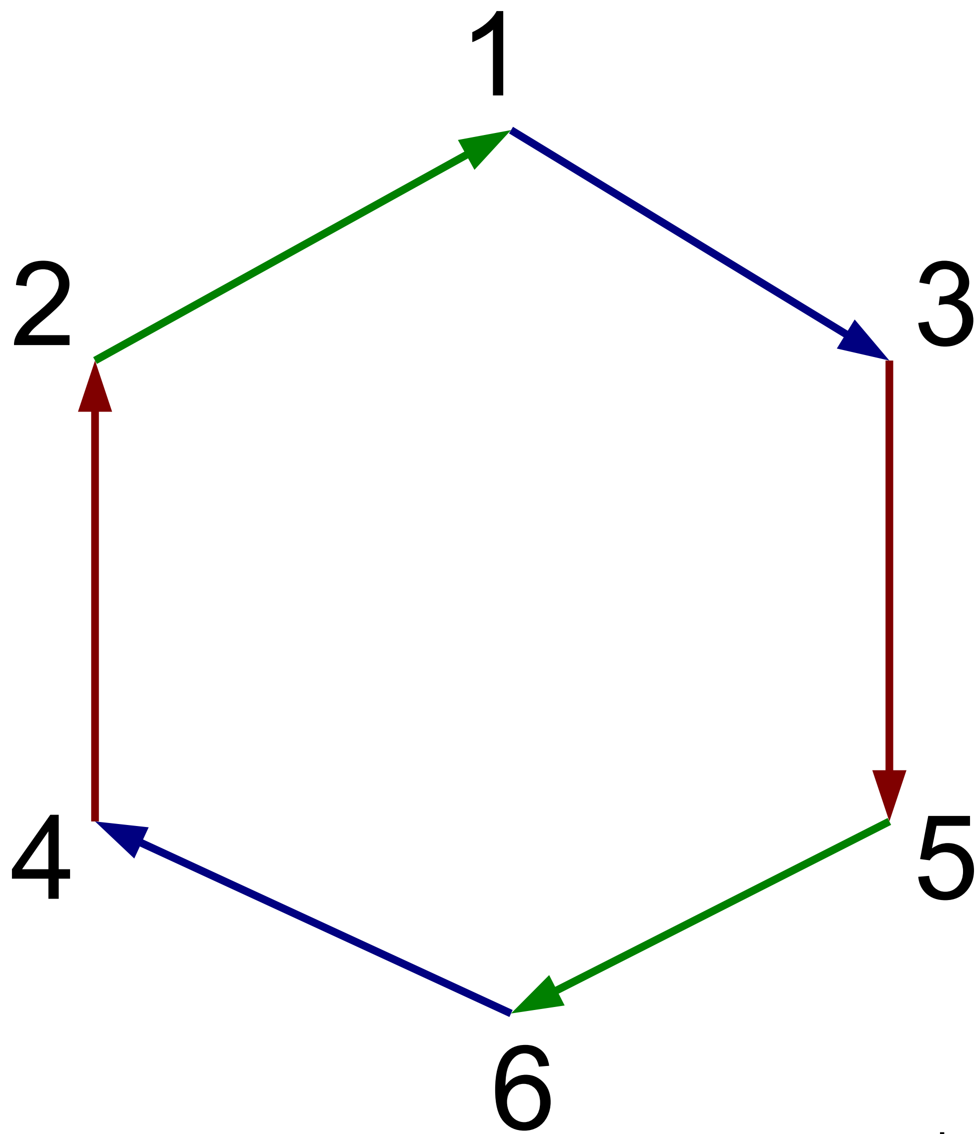


Enumerating vertices (services)



Numbering “doesn't know” anything about the loop, so it can not be done sequentially along the loop chain.

Building dependency pre-matrixes: “use”, “after” and “need”



1 →	1 → 3	1 →
2 → 1	2 →	2 →
3 →	3 →	3 → 5
4 →	4 →	4 → 2
5 → 6	5 →	5 →
6 →	6 → 4	6 →

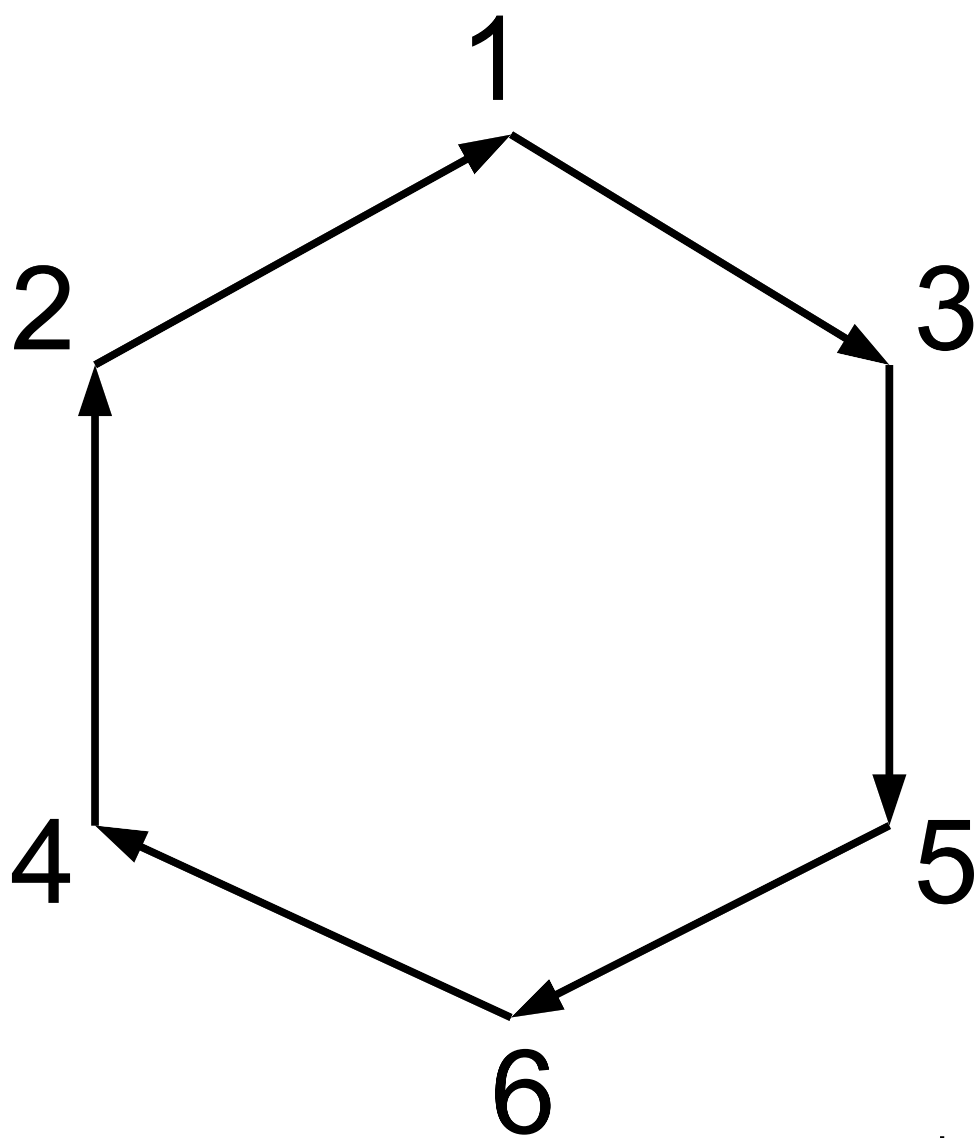
Result “use” matrix:

0						
1	1					
0						
0						
1	6					
0						

number of dependencies

dependencies

Building mixed pre-matrix for all dependency types.



1 →	1 → 3	1 →	1 → 3
2 → 1	2 →	2 →	2 → 1
3 →	3 →	3 → 5	3 → 5
4 →	4 →	4 → 2	4 → 2
5 → 6	5 →	5 →	5 → 6
6 →	6 → 4	6 →	6 → 4

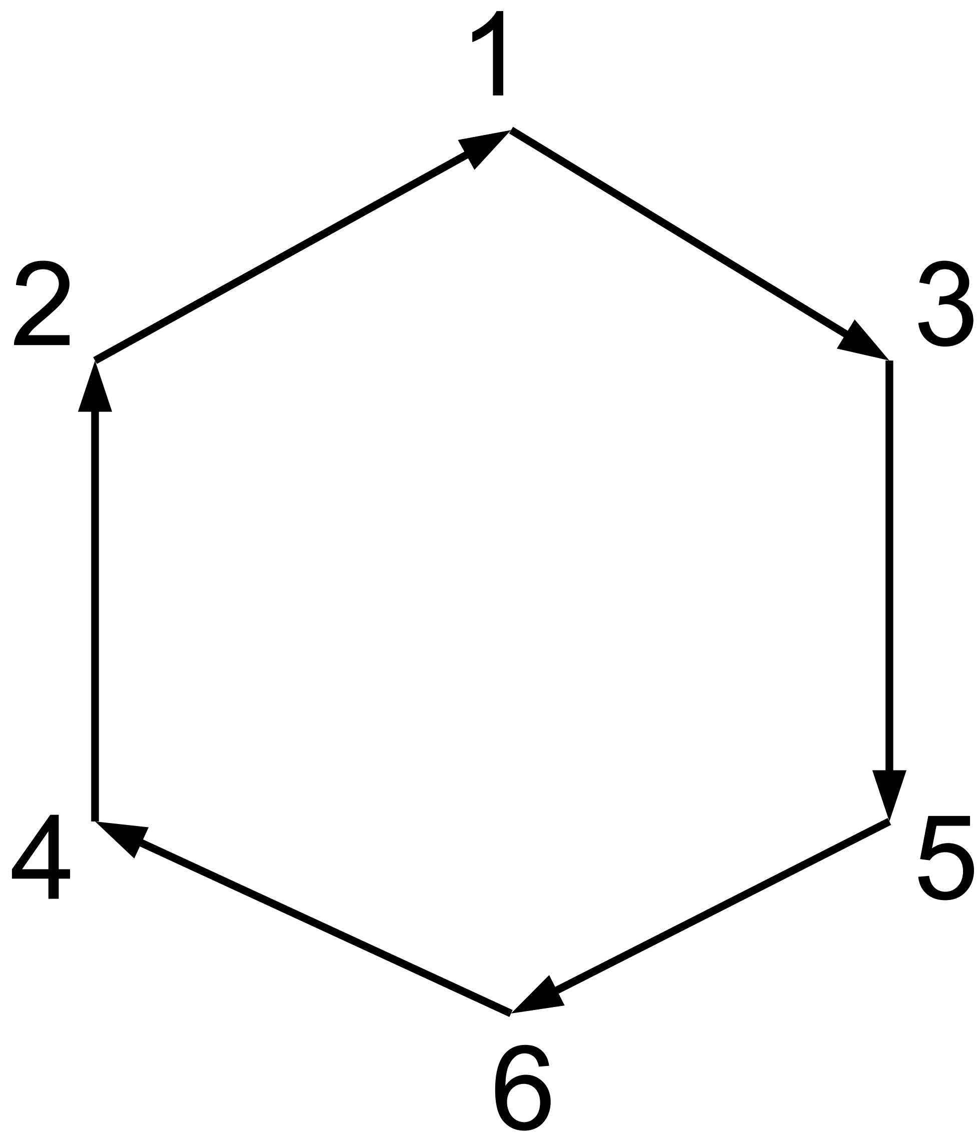
Result mixed (pre-)matrix:

1	3					
1	1					
1	5					
1	2					
1	6					
1	4					

number of dependencies

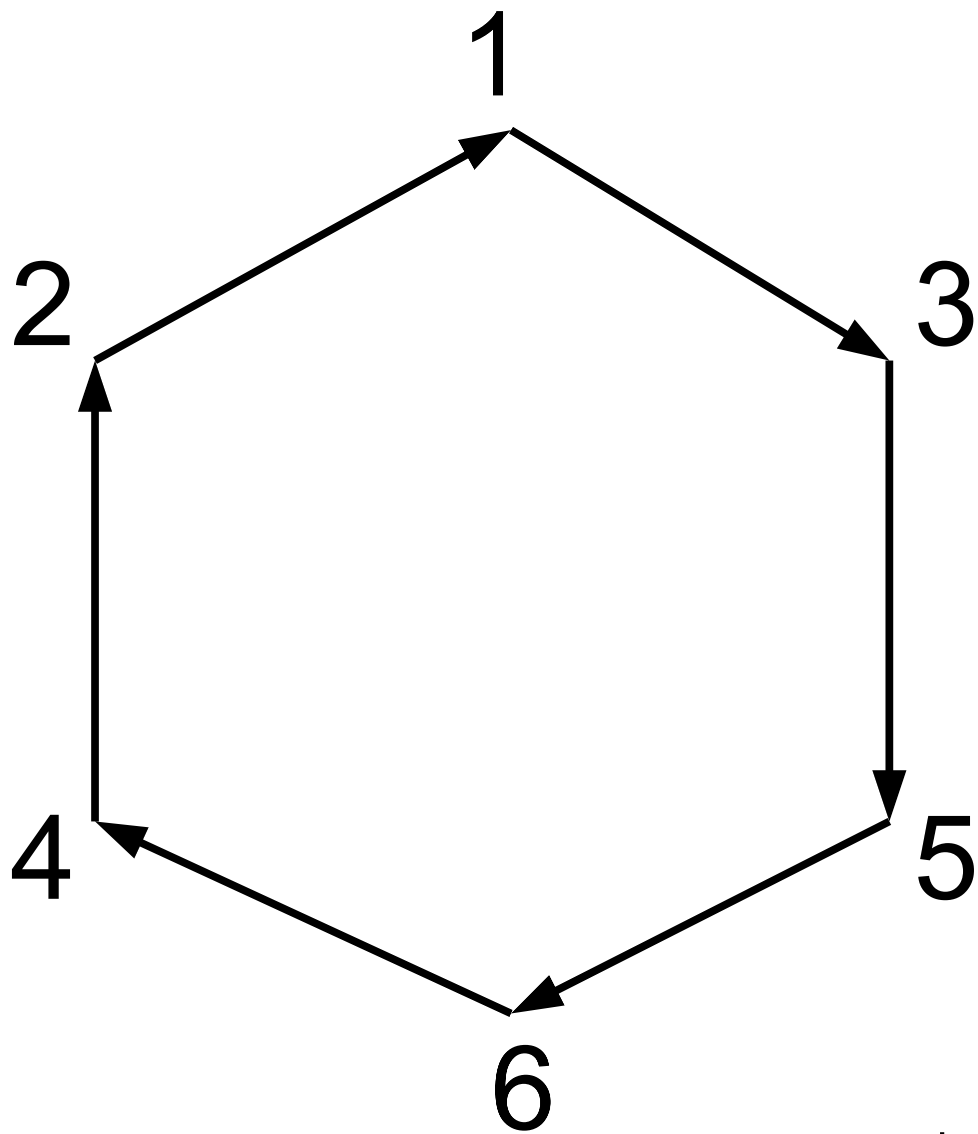
dependencies

Building matrix for all dependencies for every service (vertex)



Loop through each vertex, and look into depending dependencies, this complementing their own dependencies.

Building matrix for all dependencies for every service (vertex)



The 1-st step

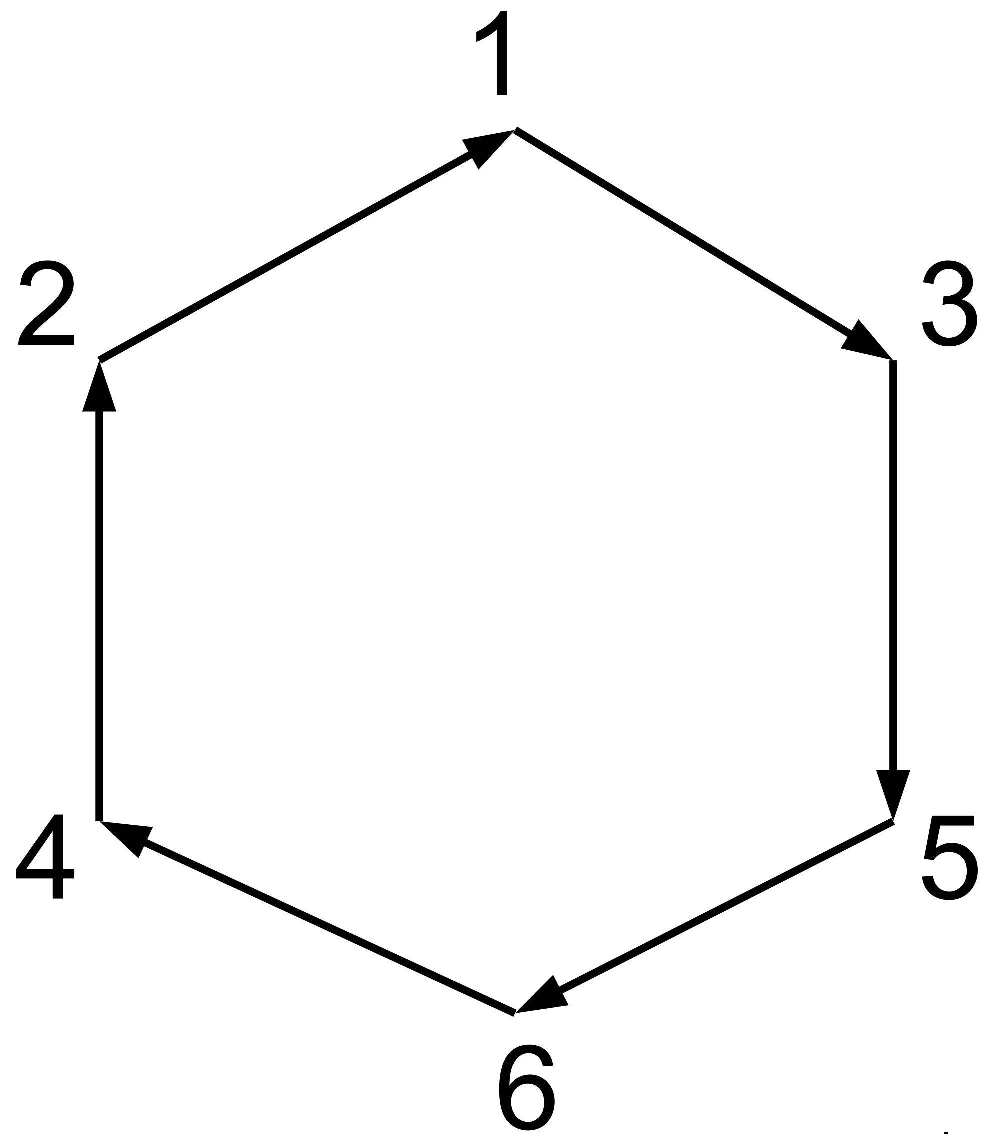
Result mixed matrix:

1	2	3	5				
2	1	1					
3	1	5					
4	1	2					
5	1	6					
6	1	4					

number of dependencies

dependencies

Building matrix for all dependencies for every service (vertex)



The 2-nd step

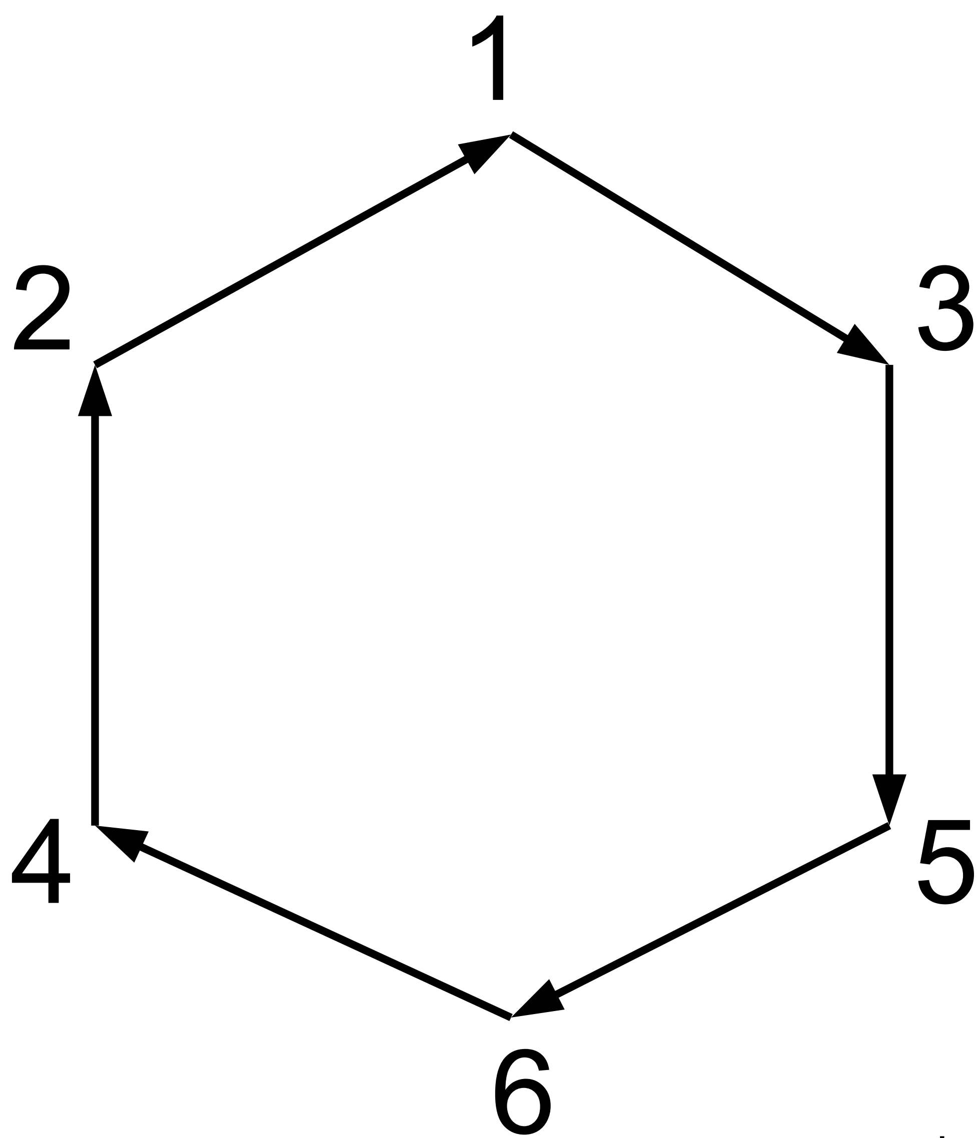
Result mixed matrix:

1	2	3	5		
2	3	1	3	5	
3	1	5			
4	1	2			
5	1	6			
6	1	4			

number of dependencies

dependencies

Building matrix for all dependencies for every service (vertex)



The 3-rd step

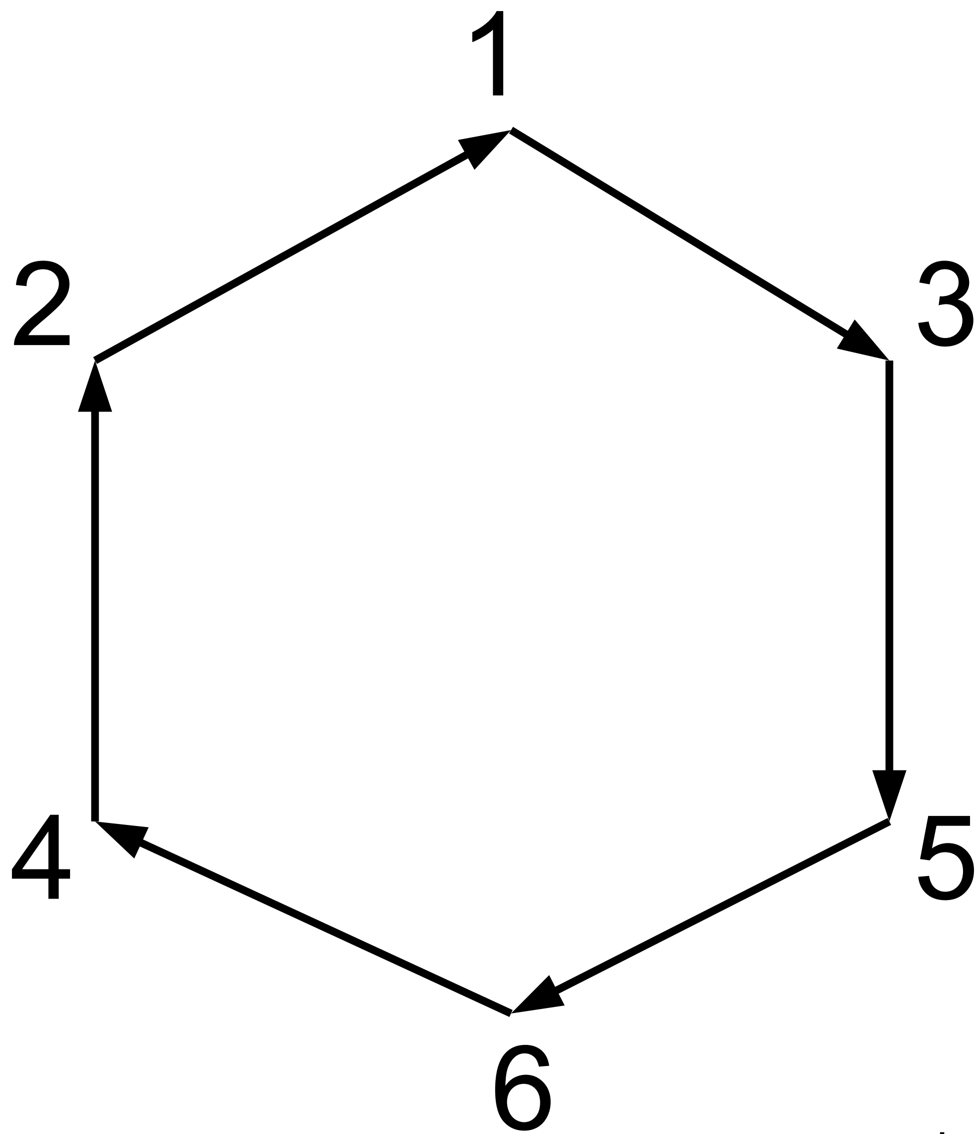
Result mixed matrix:

1	2	3	5		
2	3	1	3	5	
3	2	5	6		
4	1	2			
5	1	6			
6	1	4			

number of dependencies

dependencies

Building matrix for all dependencies for every service (vertex)



The 6-th step

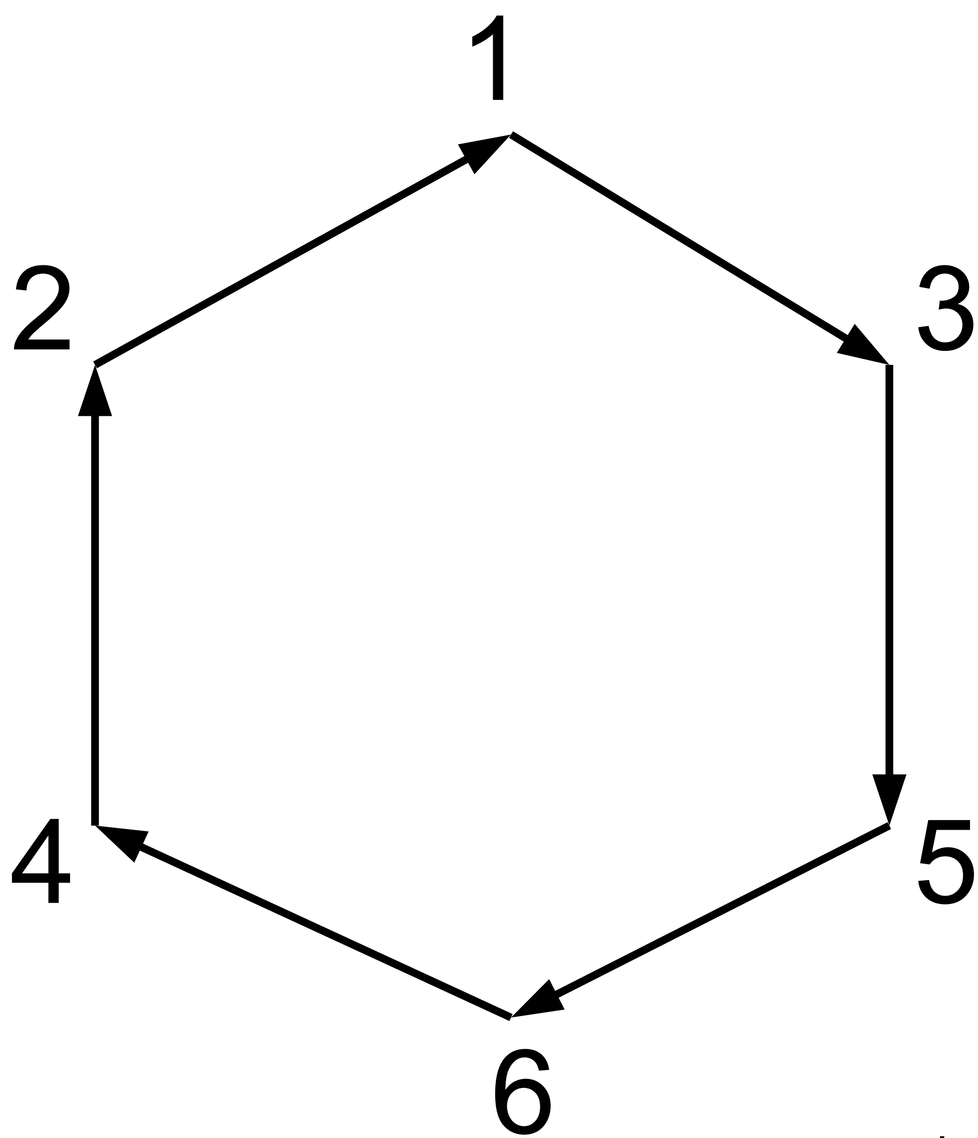
Result mixed matrix:

1	2	3	5			
2	3	1	3	5		
3	2	5	6			
4	4	2	1	3	5	
5	2	6	4			
6	5	4	2	1	3	5

number of dependencies

dependencies

Building matrix for all dependencies for every service (vertex)



As you can see the last verticles have more information about their dependencies that the first one. That's why it's more optimal to do reverse-way iteration instead of one more direct.

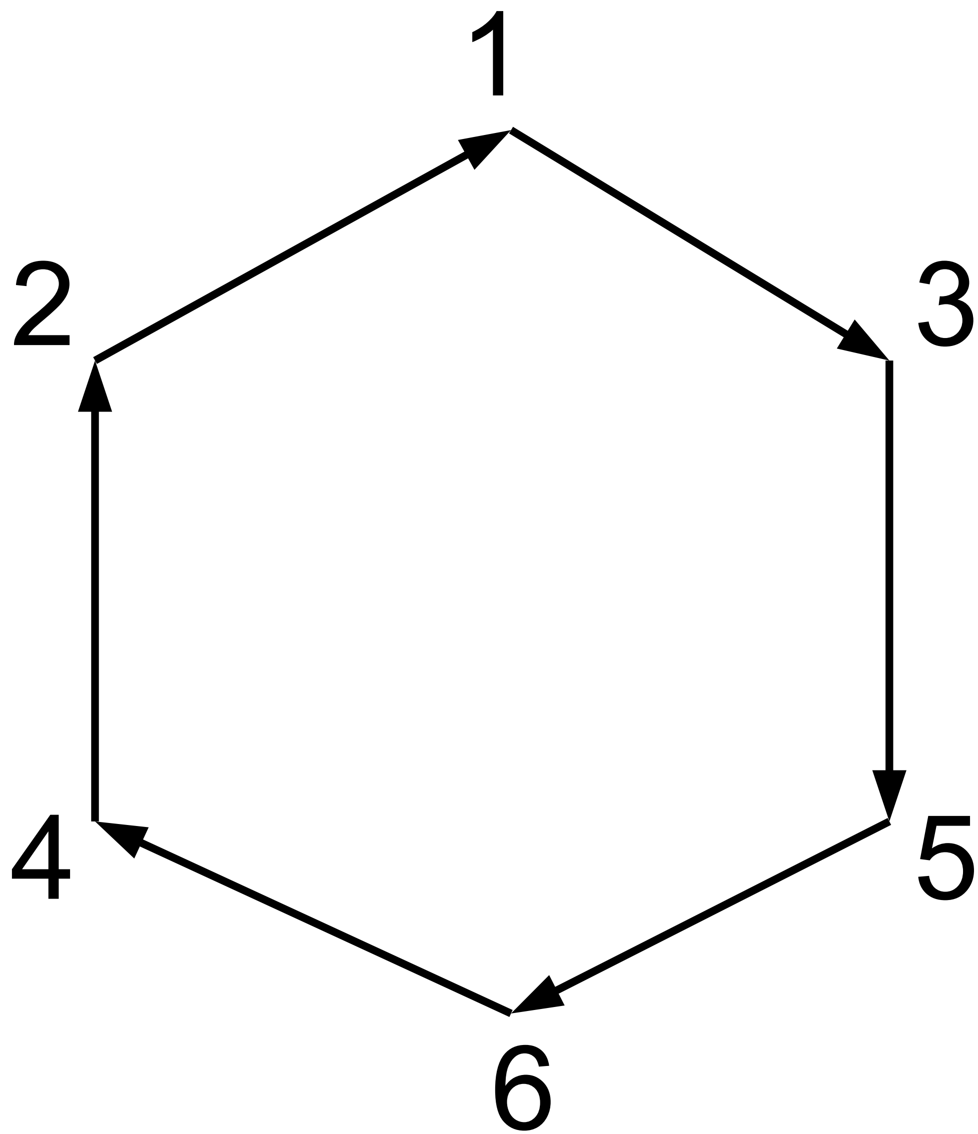
Result mixed matrix:

1	2	3	5			
2	3	1	3	5		
3	2	5	6			
4	4	2	1	3	5	
5	2	6	4			
6	5	4	2	1	3	5

number of dependencies

dependencies

Building matrix for all dependencies for every service (vertex)



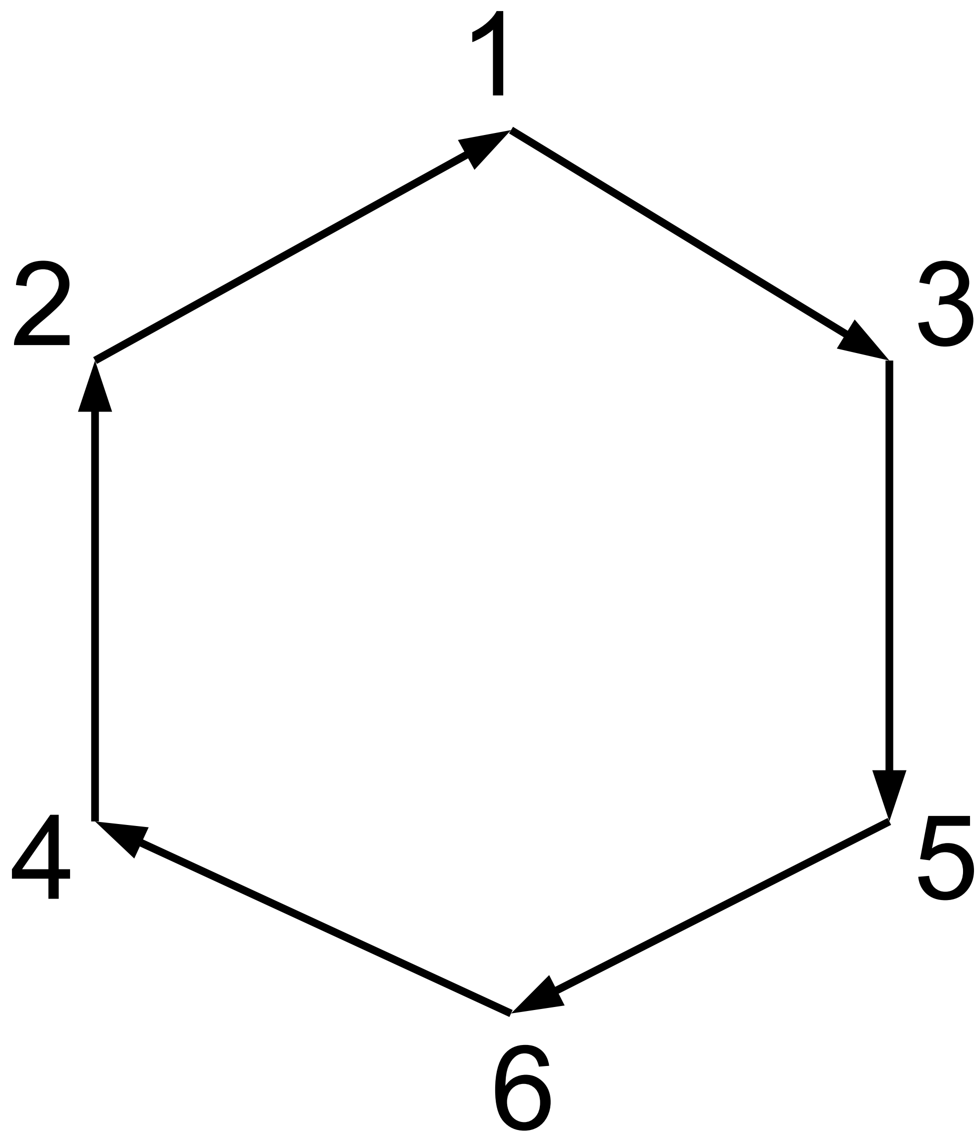
The 1-st step of the reverse-way iteration

Result mixed matrix:

1	2	3	5				
2	3	1	3	5			
3	2	5	6				
4	4	2	1	3	5		
5	2	6	4				
6	6	4	2	1	3	5	**6**

You can see a loop right here

Building matrix for all dependencies for every service (vertex)



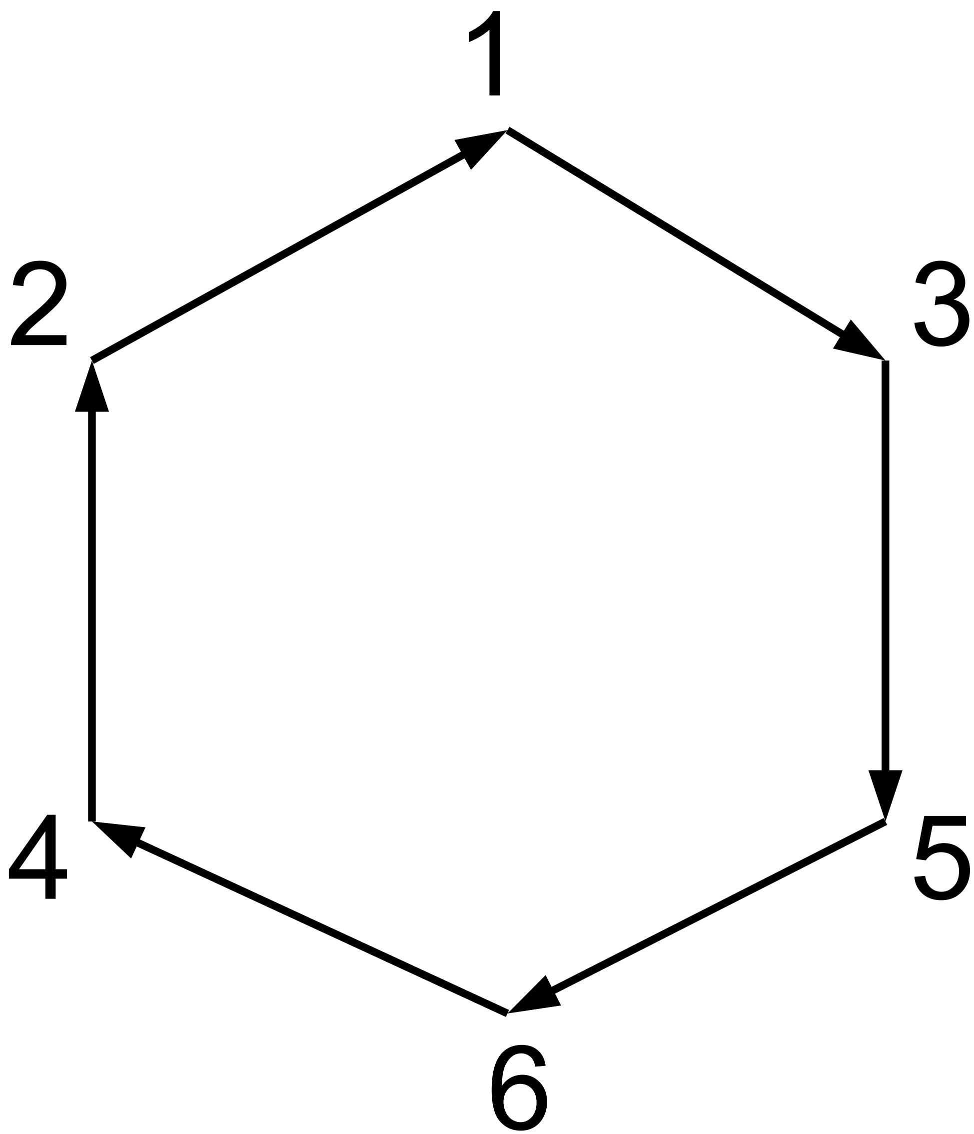
The 2-nd step of the reverse-way iteration

Result mixed matrix:

1	2	3	5				
2	3	1	3	5			
3	2	5	6				
4	4	2	1	3	5		
5	6	6	4	2	1	3	**5**
6	6	4	2	1	3	5	**6**

and here

Building matrix for all dependencies for every service (vertex)



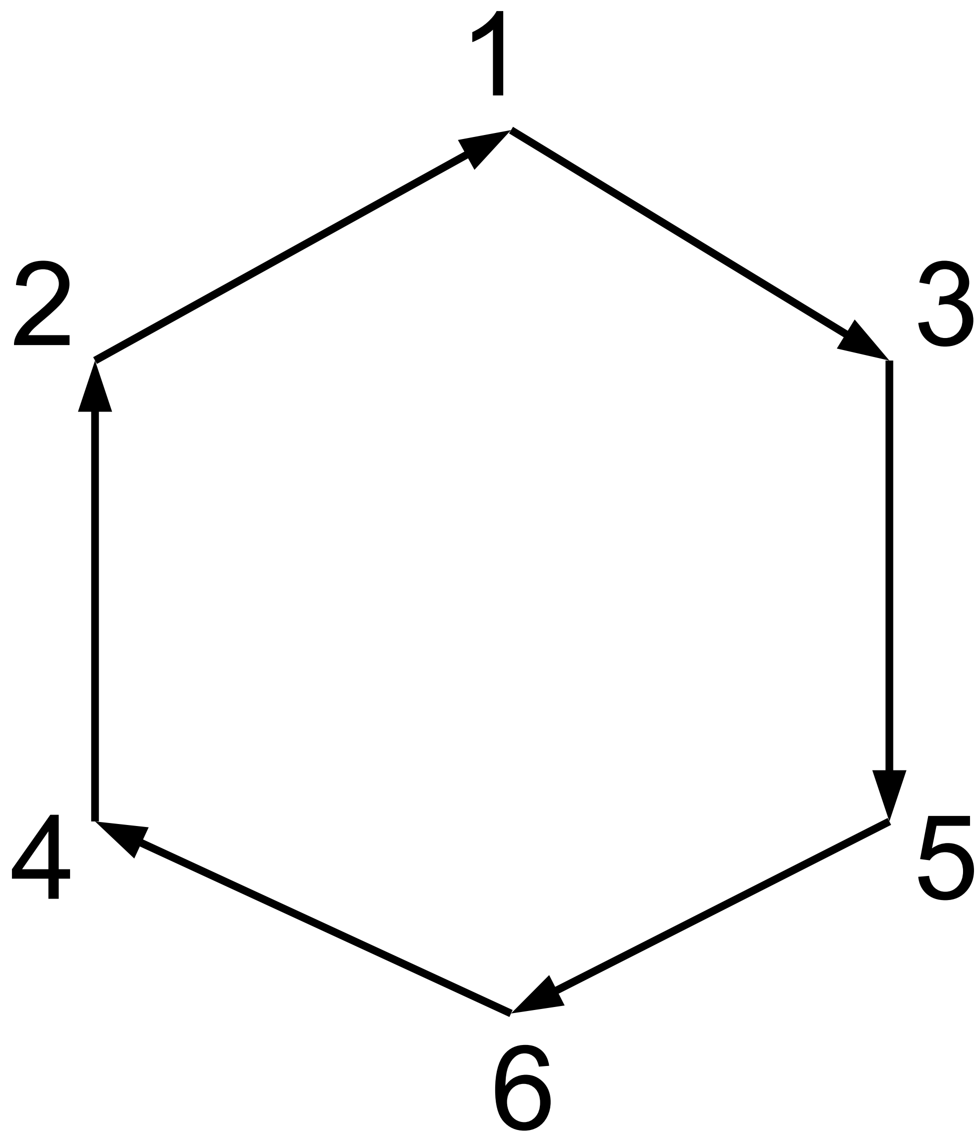
The 3-rd step of the reverse-way iteration

Result mixed matrix:

1	2	3	5				
2	3	1	3	5			
3	2	5	6				
4	6	2	1	3	5	6	4
5	6	6	4	2	1	3	**5**
6	6	4	2	1	3	5	**6**

and here...

Building matrix for all dependencies for every service (vertex)



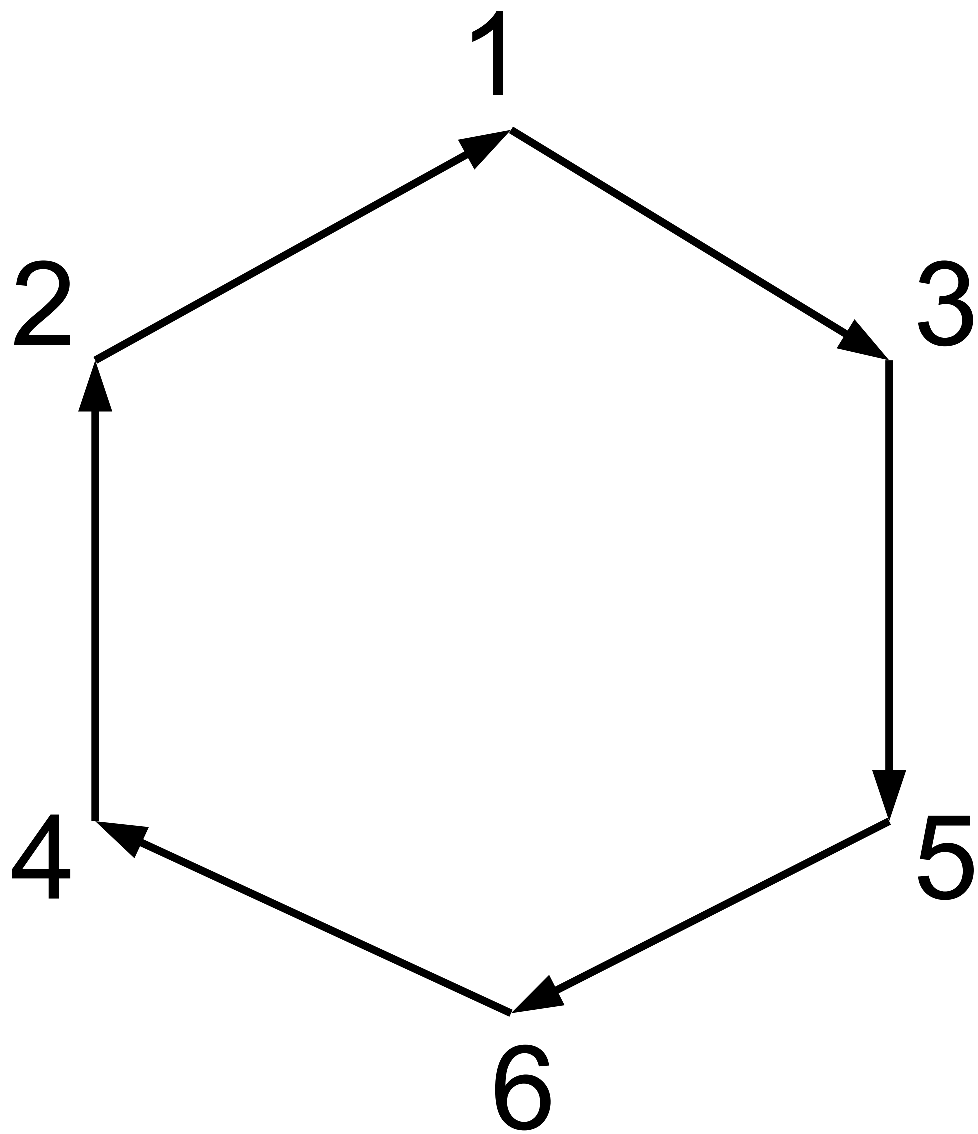
The 6-th step of the reverse-way iteration

Result mixed matrix:

1	6	3	5	6	4	2	1
2	6	1	3	5	6	4	2
3	6	5	6	4	2	1	3
4	6	2	1	3	5	6	4
5	6	6	4	2	1	3	5
6	6	4	2	1	3	5	6

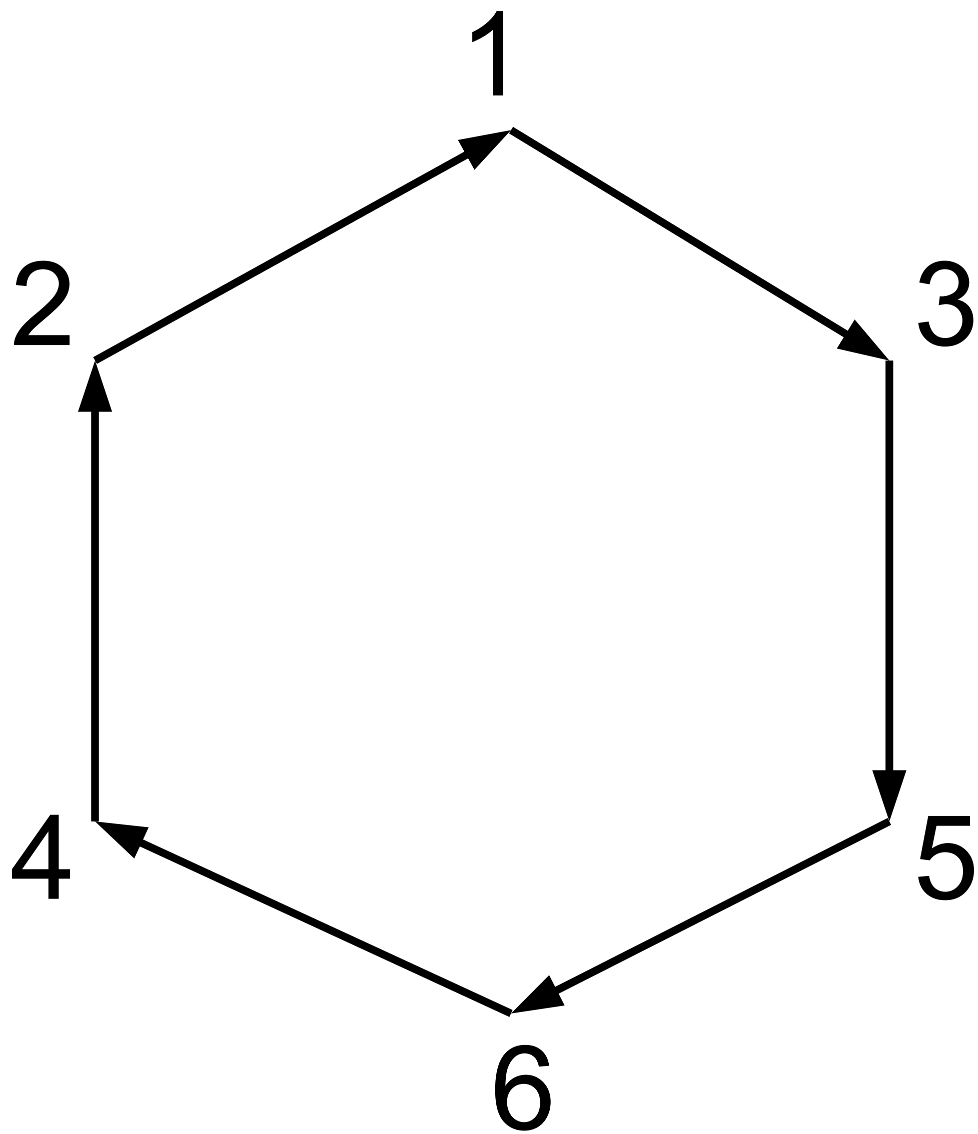
and even here

Building matrix for all dependencies for every service (vertex)



I have a hypothesis, that the only one full iteration is enough to detect any loop. But I have no prove of that. So to be sure, “direct+reverse” iterations are repeating until no modifications will be made into the matrix

Detecting the loops

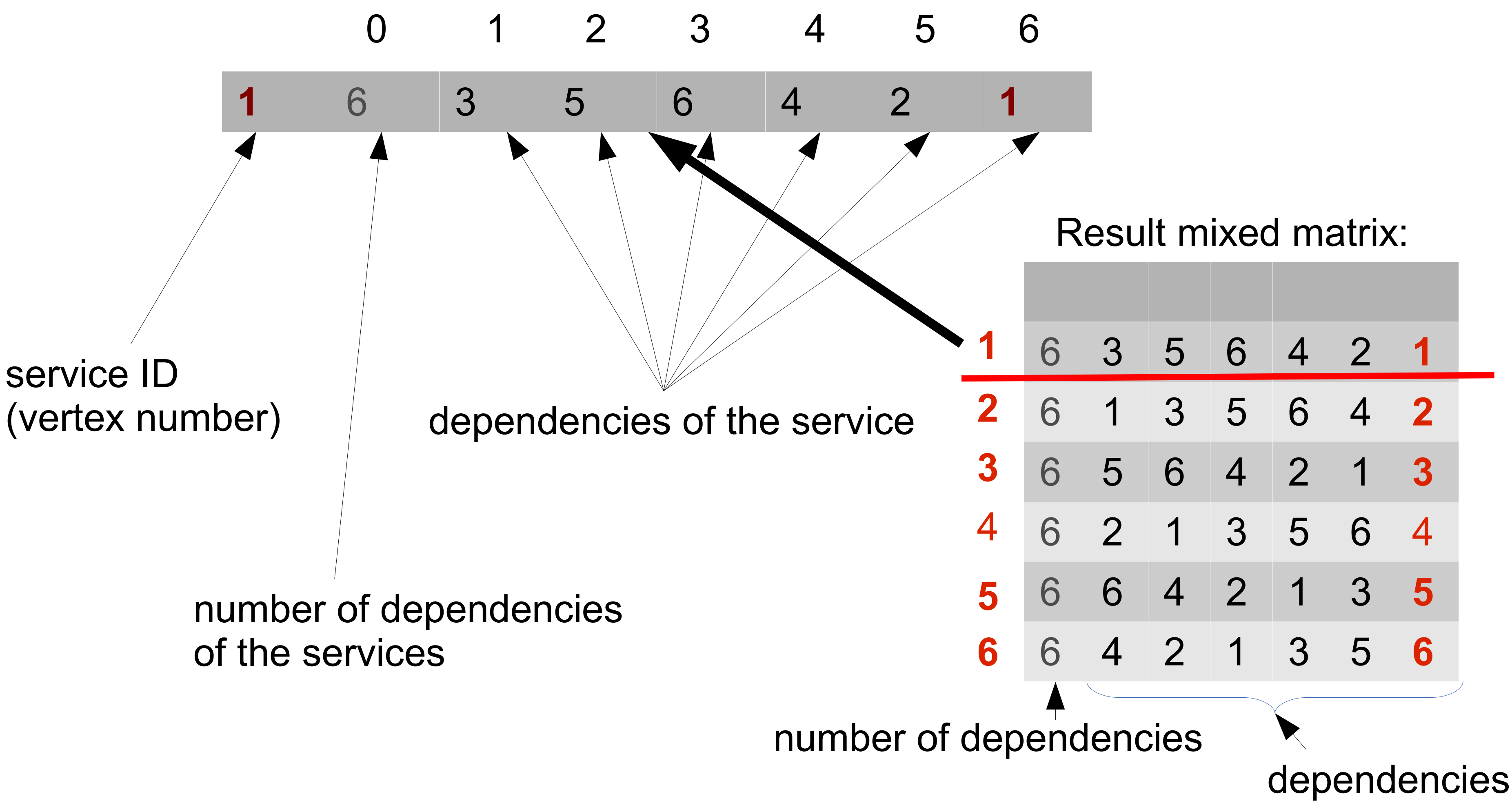


Detecting the loops after that is a very simple task. It's just need to check if a service (vertex) is depended on itself.

Result mixed matrix:

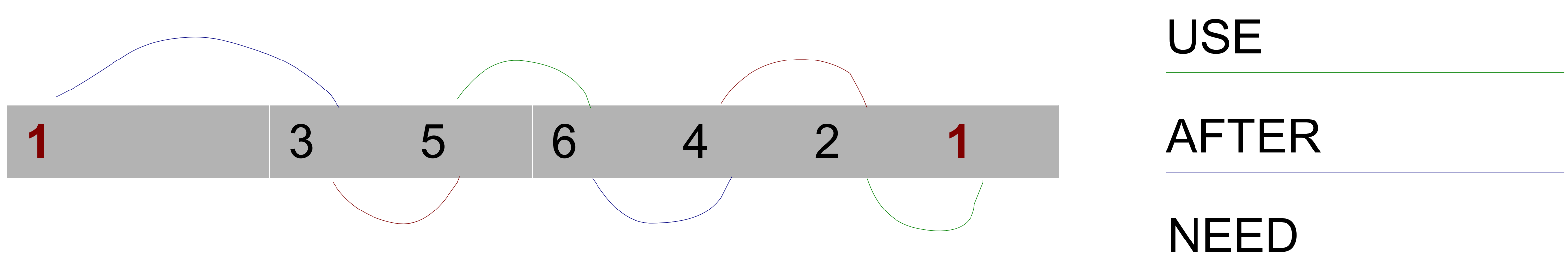
1	6	3	5	6	4	2	1
2	6	1	3	5	6	4	2
3	6	5	6	4	2	1	3
4	6	2	1	3	5	6	4
5	6	6	4	2	1	3	5
6	6	4	2	1	3	5	6

Solving the loops



Solving the loops

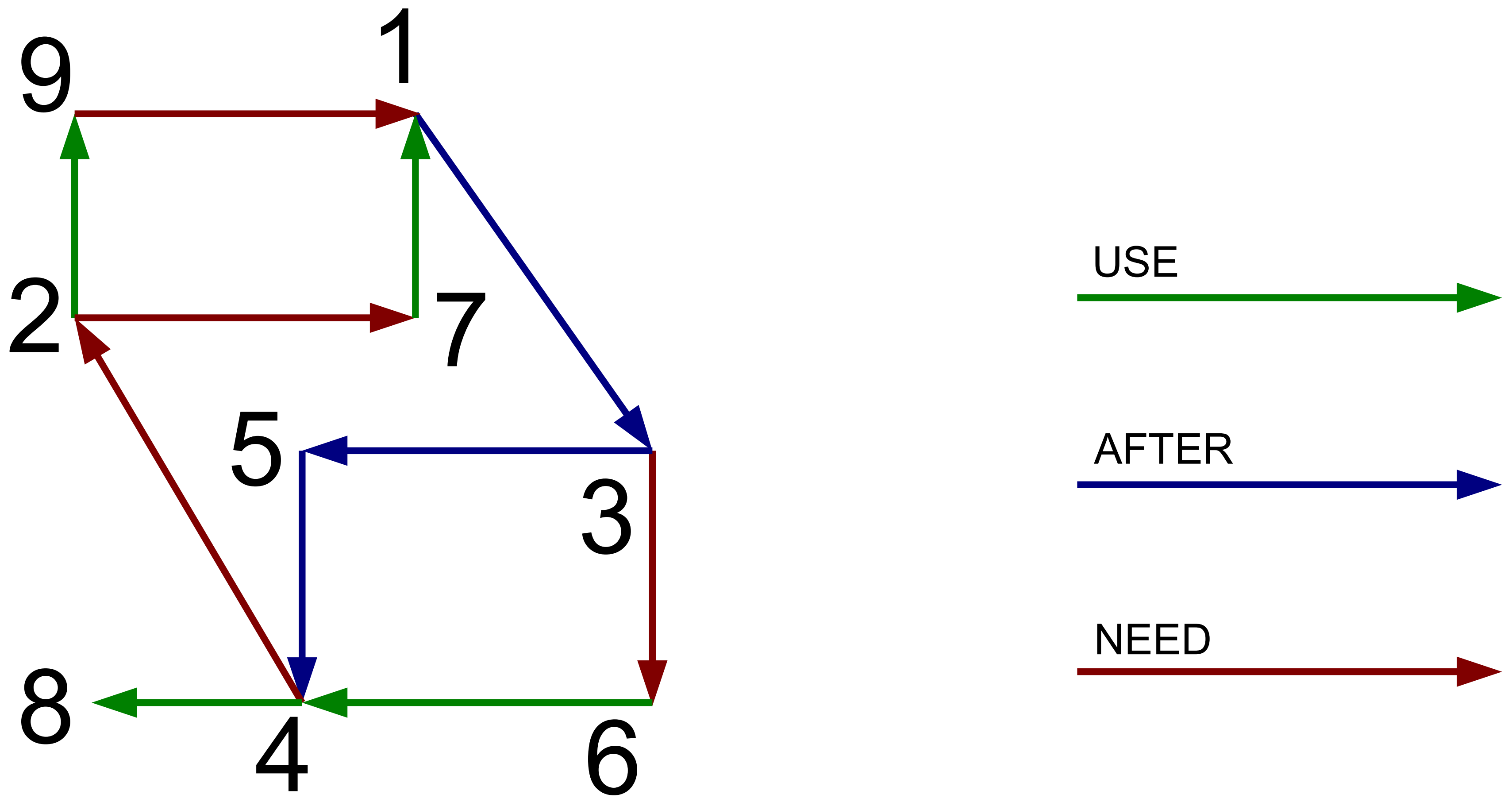
Any later dependency in a line of the matrix may be caused only by the earlier one (or by the service itself). So using pre-matrixes of dependencies of any type we can restore the picture of dependencies.



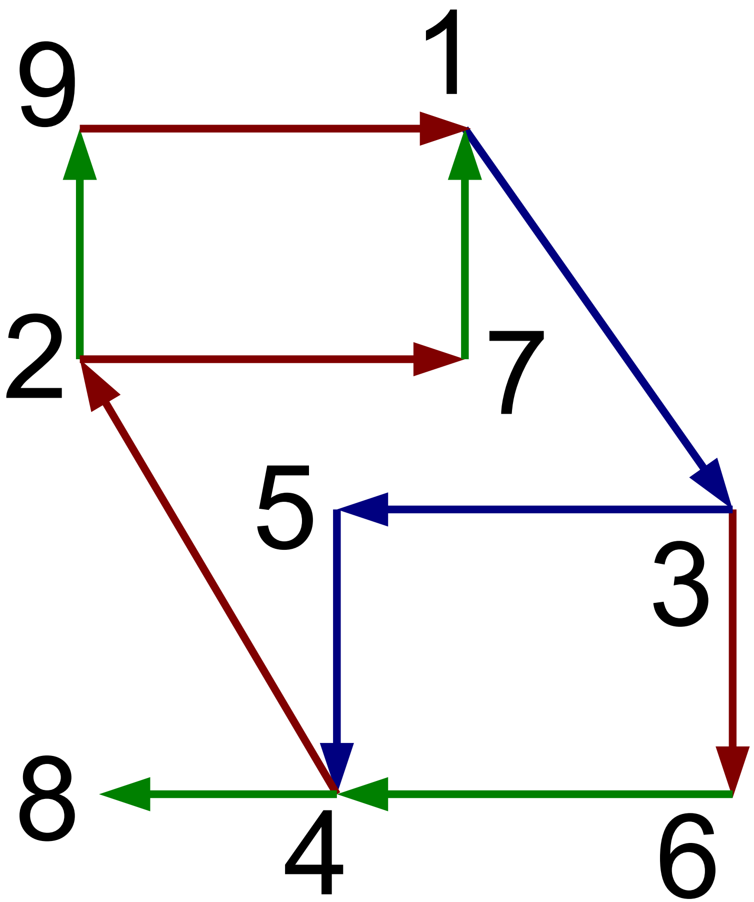
But to solve the loop of the example, it's enough just to break dependency 5 → 6. However in real case the dependency loop may be very branched, and it's required to consider with the every branch. So, to be more effective, let's descry solver's algorithm on some much more branched example.

… take a look at next example...

A branched example



A branched example



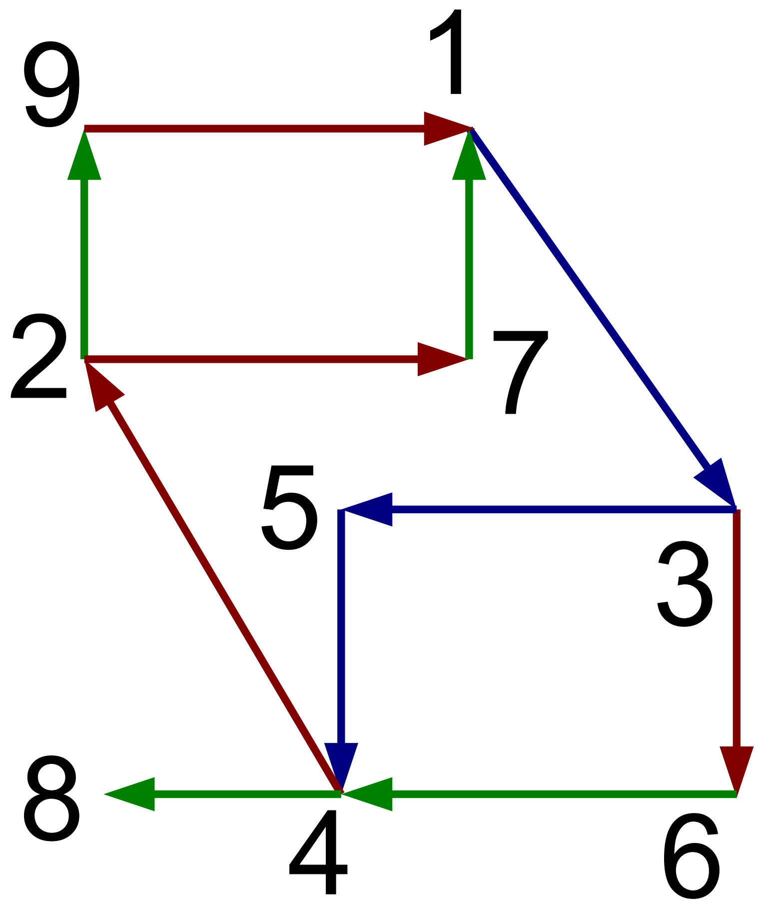
Obviously, it's recommended to break:

- 2 → 9 and
- 7 → 1.

Let's begin:

First of all, it's required to detect the loop...

Getting matrixes

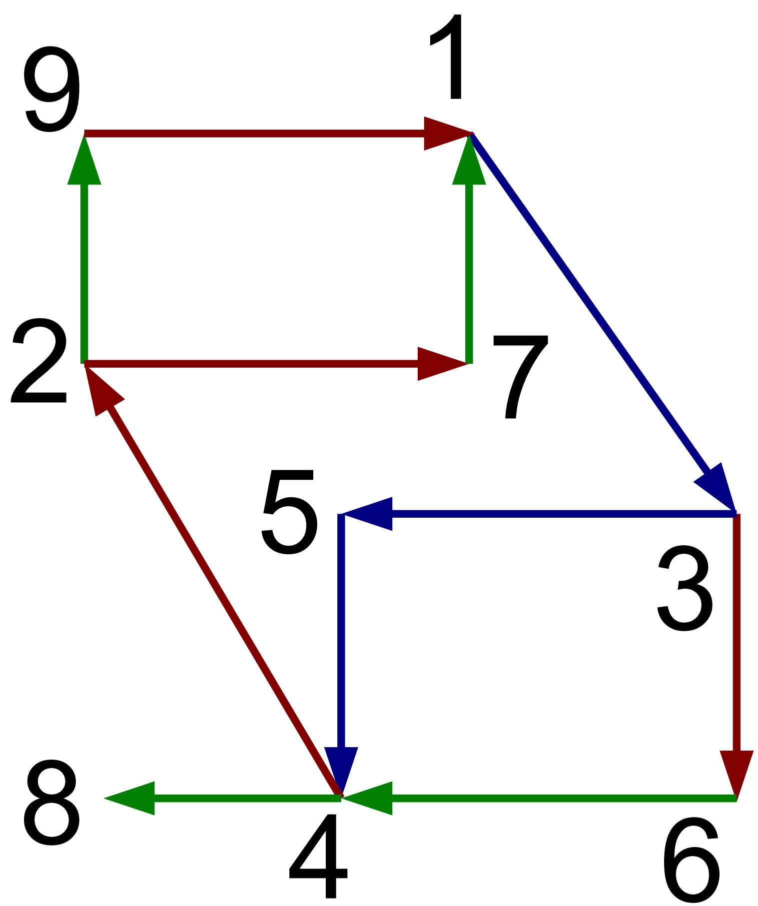


1 →	1 → 3	1 →	1 → 3
2 → 9	2 →	2 → 7	2 → 7, 9
3 →	3 →	3 → 6	3 → 5, 6
4 → 8	4 → 5	4 → 2	4 → 2, 8
5 →	5 → 4	5 →	5 → 4
6 → 4	6 →	6 →	6 → 4
7 → 1	7 →	7 →	7 → 1
8 →	8 →	8 →	8 →
9 →	9 →	9 → 1	9 → 1

Result mixed (pre-)matrix:

1	3					
2	7	9				
2	5	6				
2	2	8				
1	4					
1	4					
1	1					
0						
1	1					

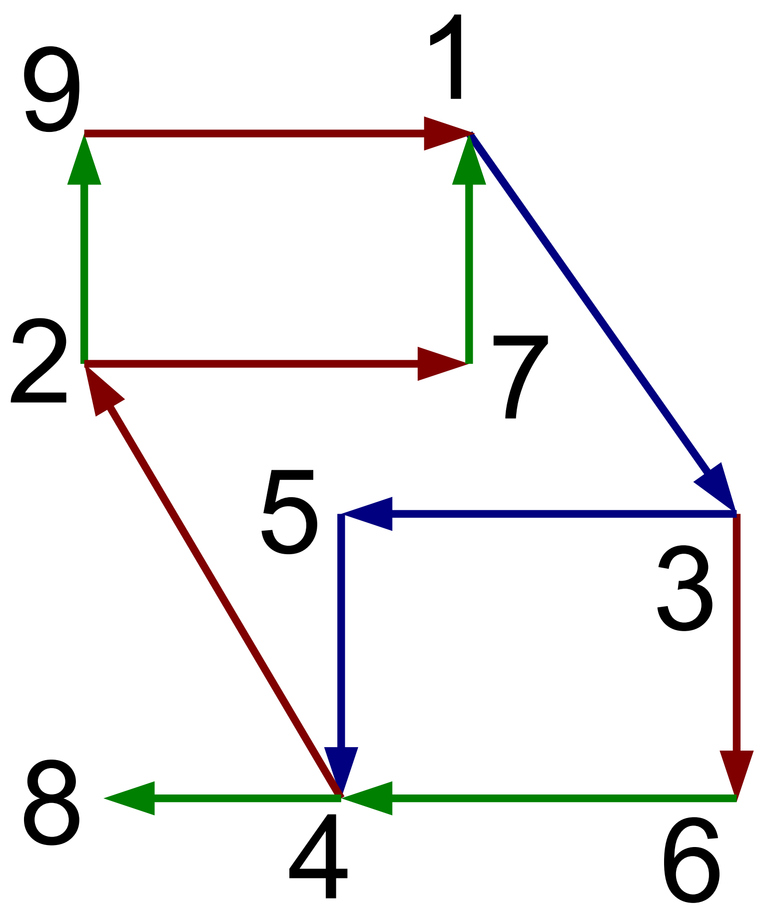
Direct way expanding



Result mixed matrix:

1	1	3					
2	3	7	9	1			
3	3	5	6	4			
4	5	2	8	7	9	1	
5	6	4	2	8	7	9	1
6	6	4	2	8	7	9	1
7	2	1	3				
8	0						
9	2	1	3				

Reverse way expanding



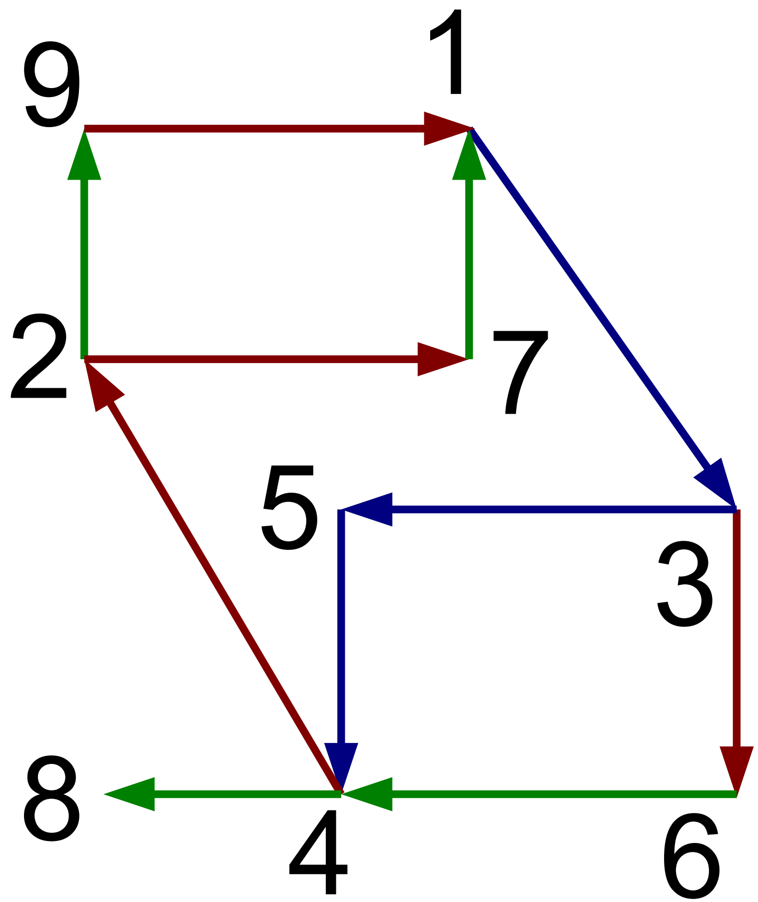
As we can see, the loop is already detected.

But we decided to repeat this operation until no matrix modifications will be done (see slide #16).

Result mixed matrix:

1	9	3	5	6	4	2	8	7	9	**1**
2	7	7	9	1	3	5	6	4		
3	9	5	6	4	2	8	7	9	1	**3**
4	9	2	8	7	9	1	3	5	6	**4**
5	9	4	2	8	7	9	1	3	**5**	6
6	9	4	2	8	7	9	1	3	5	**6**
7	5	1	3	5	6	4				
8	0									
9	5	1	3	5	6	4				

Direct+reverse again and again

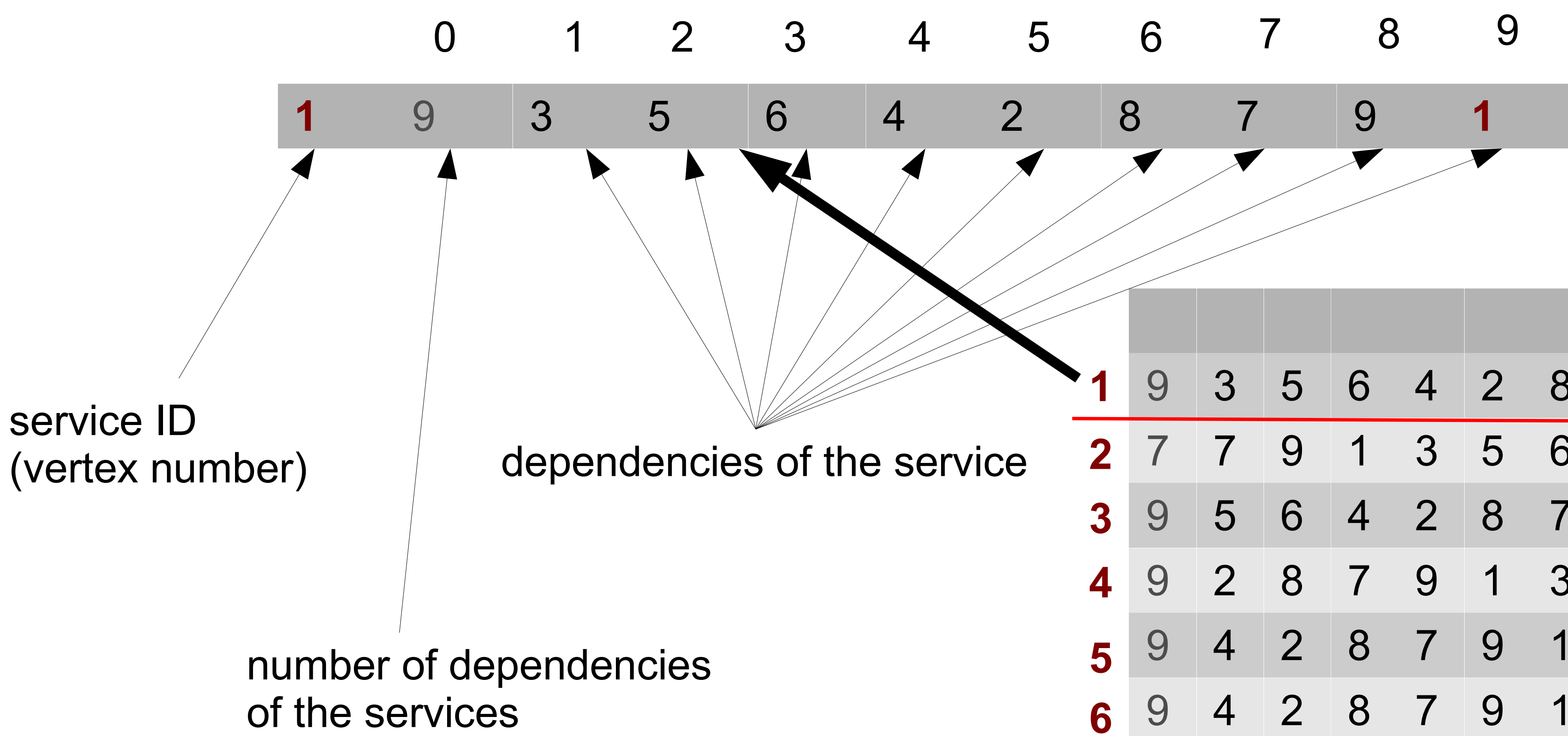


Now, OpenRC will try to solve the loop, based on chain of vertex #1

Result mixed matrix:

1	9	3	5	6	4	2	8	7	9	**1**
2	7	7	9	1	3	5	6	4	**2**	8
3	9	5	6	4	2	8	7	9	1	**3**
4	9	2	8	7	9	1	3	5	6	**4**
5	9	4	2	8	7	9	1	3	**5**	6
6	9	4	2	8	7	9	1	3	5	**6**
7	5	1	3	5	6	4	2	8	**7**	9
8	0									
9	5	1	3	5	6	4	2	8	7	**9**

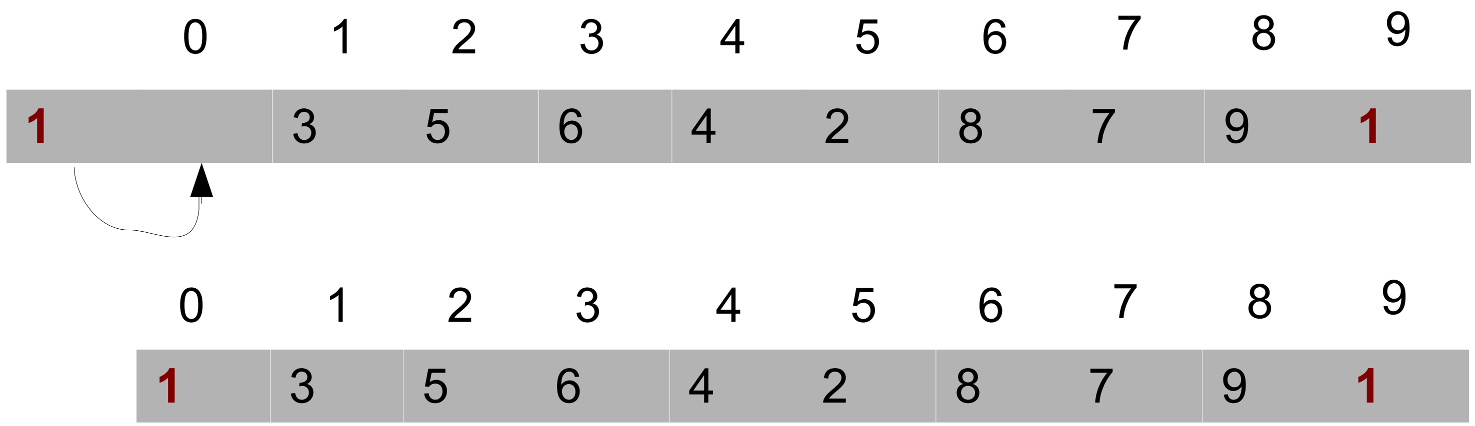
Preparing to solve the loop



1	9	3	5	6	4	2	8	7	9	**1**
2	7	7	9	1	3	5	6	4	**2**	8
3	9	5	6	4	2	8	7	9	1	**3**
4	9	2	8	7	9	1	3	5	6	**4**
5	9	4	2	8	7	9	1	3	**5**	6
6	9	4	2	8	7	9	1	3	5	**6**
7	5	1	3	5	6	4	2	8	**7**	9
8	0									
9	5	1	3	5	6	4	2	8	7	**9**

Preparing to solve the loop

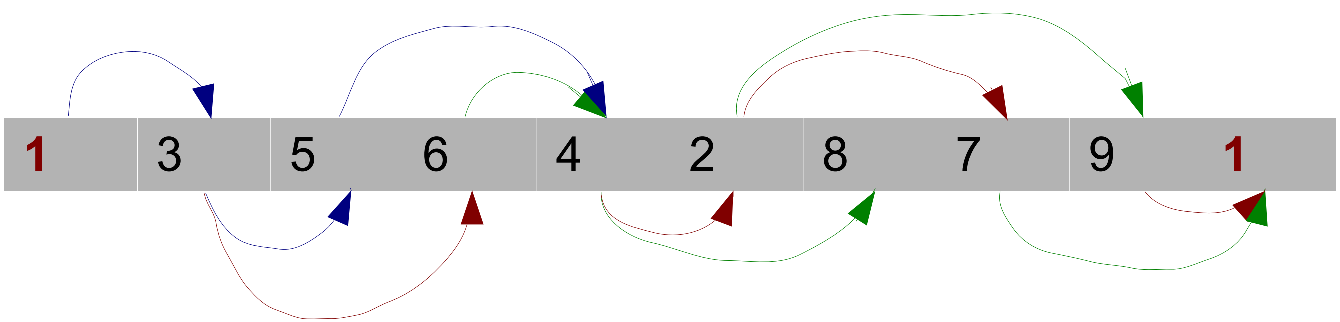
To simplify algorithm, placing service_id on the 0-th cell.



Building looping chains

And now building all possible dependency sub-chains of the chain, using dependency matrixes calculated on stage of loop detecting.

We know, that any later dependency may be caused only by some early one. So building the chains is quite simple task.



All the chains:

- 1 → 3 → 5 → 4 → 2 → 9 → 1
- 1 → 3 → 5 → 4 → 2 → 7 → 1
- 1 → 3 → 5 → 4 → 8
- 1 → 3 → 6 → 4 → 2 → 9 → 1
- 1 → 3 → 6 → 4 → 2 → 7 → 1
- 1 → 3 → 6 → 4 → 8

excluding non-looping chains

Looping chains only:

- 1 → 3 → 5 → 4 → 2 → 9 → 1
- 1 → 3 → 5 → 4 → 2 → 7 → 1
- 1 → 3 → 6 → 4 → 2 → 9 → 1
- 1 → 3 → 6 → 4 → 2 → 7 → 1

Calculating the optimal way to solve the loop

Step 1: Checking if the loop can be solved with breaking:

- only “use” dependencies,
- “use” and “after”.

or cannot be solved with breaking “use”/”after” dependencies (there is a chain that is fully consists of “need” dependencies). Don't try to solve if there's at least one unsolvable chain (end of solver).

Step 2: Counting each (only “use” or “use/after”, depending on result of step #1) dependency through each chain. It's required to determine a way with minimal dependency breaks to solve the loop.

Step 3: Removing the most present dependency through all chains.

Step 4: Checking if loop situation is solved and return to step #3 if not.

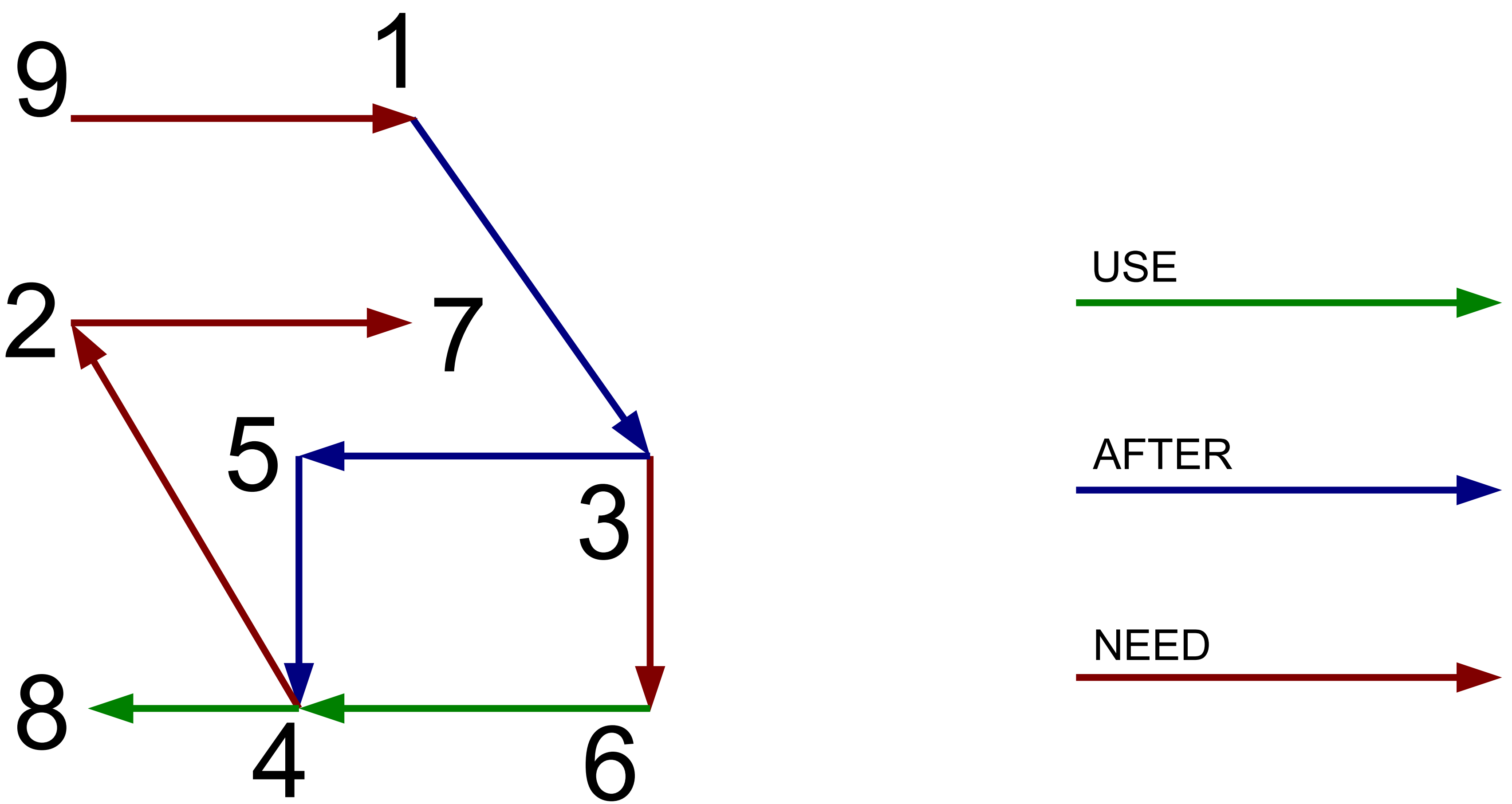
In this example, broken dependencies will be:

- 2 → 9
- 7 → 1

Looping chains:

- 1 → 3 → 5 → 4 → 2 → 9 → 1
- 1 → 3 → 5 → 4 → 2 → 7 → 1
- 1 → 3 → 6 → 4 → 2 → 9 → 1
- 1 → 3 → 6 → 4 → 2 → 7 → 1

The result



Notes about the solver

- As you can notice, the algorithm of the solver is not ideal and may extra cut some low-cost dependencies (use not optimal way to solve the loop). For example, it may cut 6 → 4, before cutting 2 → 9 and 7 → 1, because 6 → 4 is met in two chains as 2 → 9 and 7 → 1 are. This should be fixed/improved in the future (step 2 and 3 on the previous slide).

- On step #4 it'd be better to return to step #2 (instead of #3), but I have no time to modify that, yet.