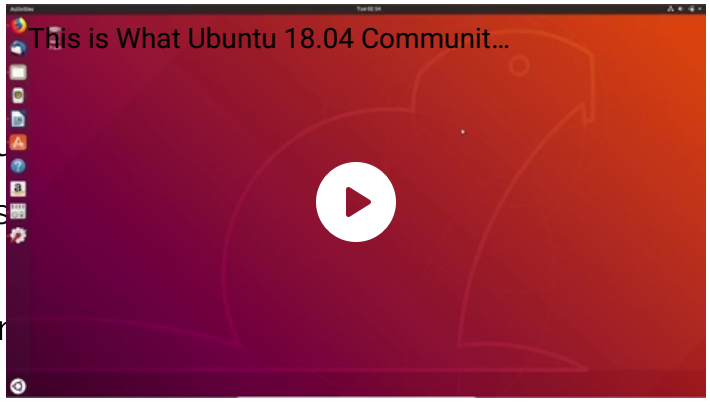
Draw a Graph Using LaTeX

Last modified: April 17, 2022

by Gabriele De Luca (https://www.baeldung.com/cs/author/gabrieledeluca)

Latex (https://www.baeldung.com/cs/category/latex)

1. Overview



In this tutorial, we'll discuss how to draw a graph u

We'll first start by listing the main LaTeX packages express their particular advantages.

Then, we'll study some examples of graphs drawn variations.

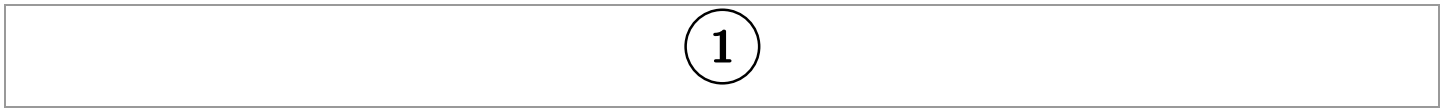
At the end of this tutorial, we'll be able to implement a graph in a LaTeX document.

2. A Short Recap on Graphs

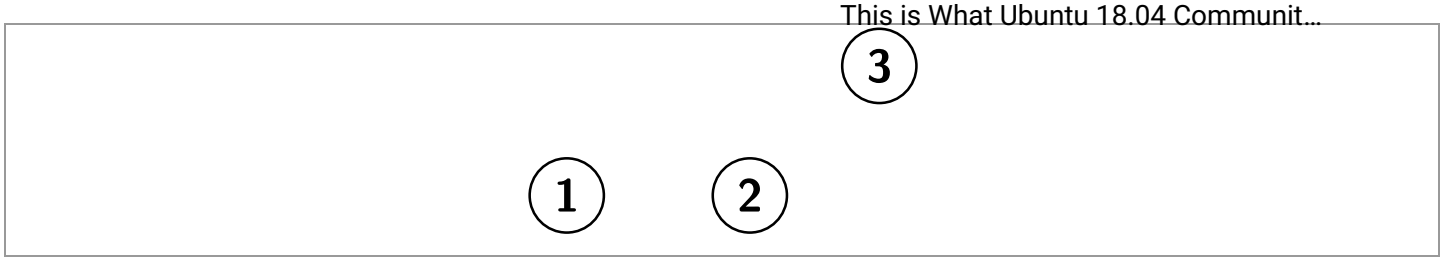
2.1. Components of Graphs and Their Representation

A graph (/cs/graph-theory-intro) $G(V, E)$ comprises a set V of vertices and a set E of edges. The edges can be either directed or undirected (/cs/graphs-directed-vs-undirected-graph), and normally connect two vertices, not necessarily distinct. For hypergraphs (https://en.wikipedia.org/wiki/Hypergraph), edges can also connect more than two edges, but we won't treat them here.

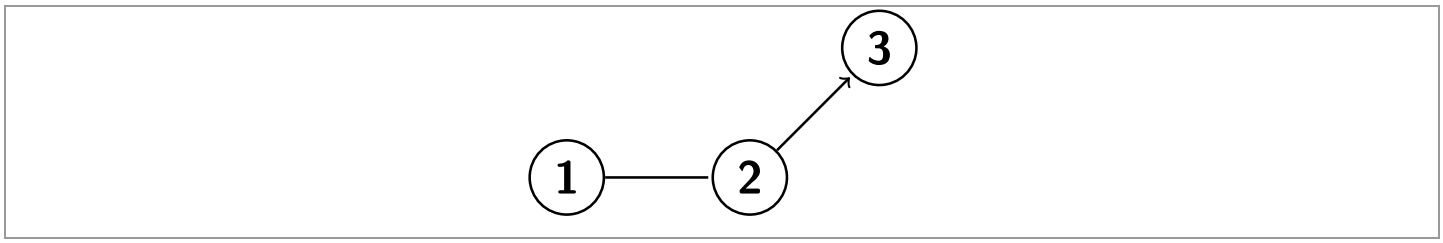
The standard representation of a vertex consists of the image of a circle:



If the vertex has a label, this is typed inside the circle, as is the case with the number 1 in the example above. We can also represent multiple vertices by placing them in a non-overlapping manner anywhere in the image:

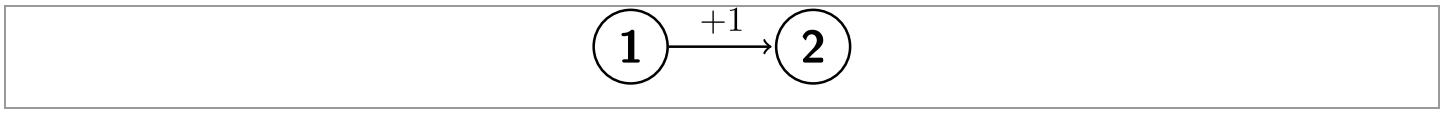


The standard representation of an edge corresponds to a line for undirected edges or an arrow for directed edges:

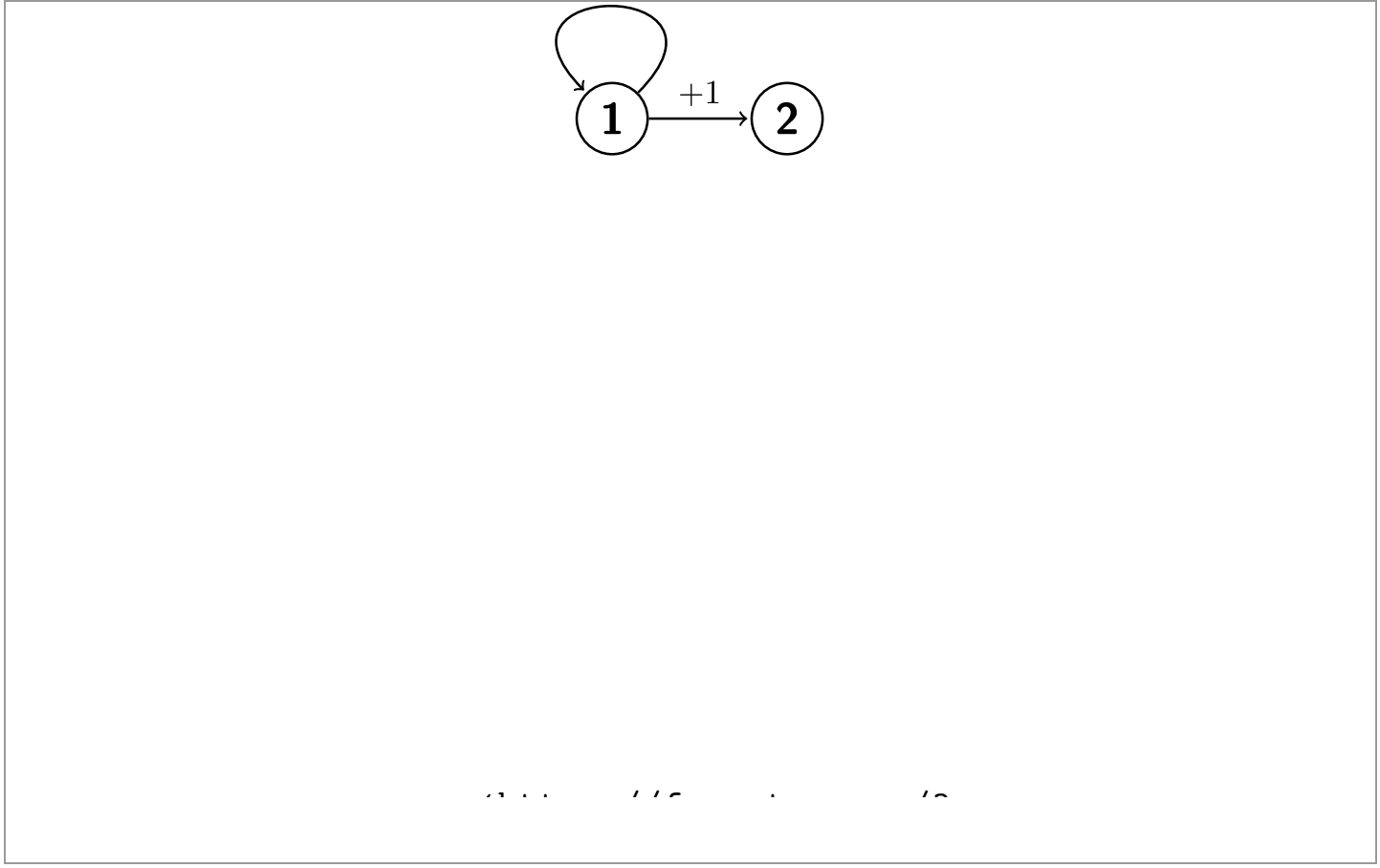


2.2. Typical Representation of Graphs

An image that represents a graph, therefore, consists of **a set of circles on an empty field and a set of lines or arrows connecting them**. Further, if the graph is a weighted graph (/java-graphs#2-weighted-graph), we can indicate the weights as labels on the edges:



And lastly, if the graph has loops, we can represent them as edges that connect a vertex to itself:



These are all the elements we need to represent a graph of any complexity (/cs/graphs#complexity). In the next section, we'll first study the relevant packages for drawing graphs, and then see some examples of the implementation of graphs in code.

2.3. Alternative Representations of Graphs

We should also state, though, that **graphical representations aren't the only way to describe graphs intuitively**. Other representations, and in particular edge lists (/cs/graphs#3-edges-list), adjacency lists (/java-graphs#2-adjacency-list), or adjacency matrices (/java-graphs#1-adjacency-matrix), are also prevalent in practice.

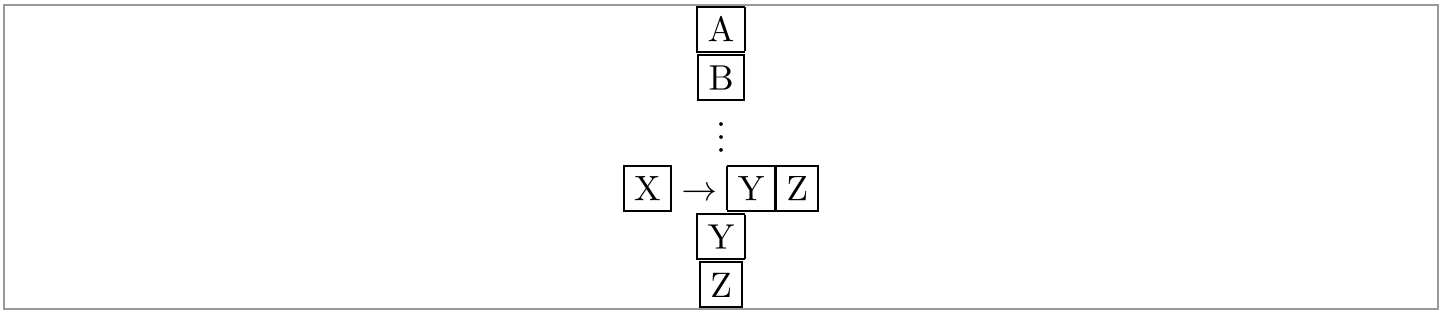
In some cases, they may be more informative to a human reader than a drawn image of a graph. For example, a directed graph with 26 vertices $V = \{A, B, ..., X, Y, Z\}$ and only two edges $E = \{(X, Y), (X, Z)\}$, is very simple to represent with formal notation:

$$G(V, \{(X, Y), (X, Z)\})$$

Its full graphical representation would however occupy a significantly vast space and be mostly uninformative. This is because, **if the graph we're using possesses more than a handful of nodes or edges, it then becomes challenging to understand its structure by looking at its image**. Some extensive graphs, such as the graph representing internet nodes (https://ieeexplore.ieee.org/document/6418770), possess

graphical representations that, while very beautiful (https://en.wikipedia.org/wiki/Internet_backbone), are however meaningless to a human.

For this reason, **if our graph holds more than a few vertices or edges, we should avoid drawing it**. Instead, we should use other types of formal representations such as adjacency matrices, edge lists, or adjacency lists:



3. Tikz

We can now discuss the packages that we can use to draw graphs in LaTeX. **The most common LaTeX package used for drawing, in general, is TikZ (https://en.wikipedia.org/wiki/PGF/TikZ)**, which is a layer over PGF (https://github.com/pgf-tikz/pgf) that simplifies its syntax. TikZ is a powerful package that comes with several libraries dedicated to specific tasks, such as:

- **Drawing mindmaps (https://github.com/pgf-tikz/pgf/blob/9ac32f11fd2f8255a8a74b0994cb9aed5c42a3e7/doc/generic/pgf/text-en/pgfmanual-en-library-mindmaps.tex)**, useful for depicting conceptual relationships
- **Diagrams for entity-relationships (https://github.com/pgf-tikz/pgf/blob/03aa54d26fe2a1d6130c65240d41262961005d86/doc/generic/pgf/text-en/pgfmanual-en-library-er.tex)**, which can assist us in **representing knowledge graphs (/cs/ml-knowledge-graphs)**
- Drawing **logical circuits (https://github.com/pgf-tikz/pgf/blob/03aa54d26fe2a1d6130c65240d41262961005d86/doc/generic/pgf/text-en/pgfmanual-en-library-circuits.tex)**, that are use[illegible] to represent operations in **Boolean algebra (/cs/boolean-algebra-basic-laws)**

And also, relevant for our purposes, it contains a *graphdrawing (https://github.com/pgf-tikz/pgf/blob/03aa54d26fe2a1d6130c65240d41262961005d86/doc/generic/pgf/text-en/pgfmanual-en-qd-usage-tikztex)* library that we can use for the automatic drawing of graphs. TikZ is the standard package we use to draw graphs, and we'll dedicate it to most of the coded examples.

3.1. Basic Graph and Nodes

Drawings in TikZ are included in the *tikzpicture* environment. The simplest graph we can draw comprises of one vertex with its label, represented as a circle containing, for example, a variable:

```
\documentclass{article}
\usepackage{tikz}
\begin{document}
\begin{tikzpicture}[main/.style = {draw, circle}]
\node[main] (1) {$x_1$};
\end{tikzpicture}
\end{document}
```

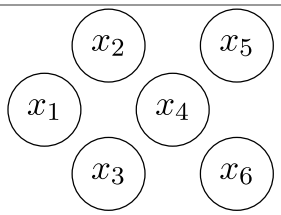
The word *main* here refers to a style that we define when introducing the *tikzpicture* environment. In this case, we use it to tell the compiler to draw the shape of a circle for the nodes.

The most important command in the code above is *\node*. Its syntax is:

```
\node[parameters] (nodeID) {nodeLabel};
```

We can use this command repeatedly to add more vertices to the graph:

```
\node[main] (2) [above right of=1] {$x_2$};
\node[main] (3) [below right of=1] {$x_3$};
\node[main] (4) [above right of=3] {$x_4$};
\node[main] (5) [above right of=4] {$x_5$};
\node[main] (6) [below right of=4] {$x_6$};
```



Notice how the term between the second pair of square brackets lets us define the position of the nodes in relation to other nodes. We can do this by using the keywords *right*, *left*, *above*, *below*, followed by a blank space, and then the keyword *of* and the label of the node to which they refer. This is particularly useful if we later need to rearrange the graph on the picture. In fact, by working with relative positions as opposed to absolute coordinates, we can move one anchor node around to move all others.

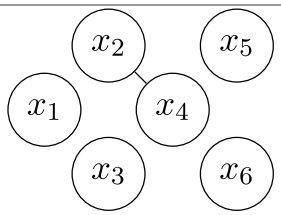
3.2. Adding Edges

We can also add edges by using the *\draw* command:


```
\draw[parameters] (fromWhere) -- (toWhere);
```

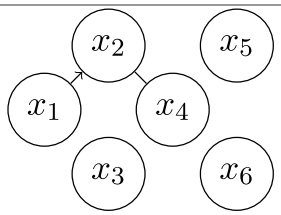
If we, for example, want to draw a line corresponding to an undirected edge between nodes (2) and (4), we can write:

```
\draw (2) -- (4);
```



If instead, we want to add an arrow for a directed edge, we can give the parameter `->` to the *\draw* command:

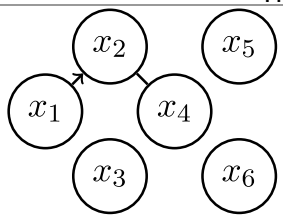
```
\draw[->] (1) -- (2);
```



3.3. Distances and Widths

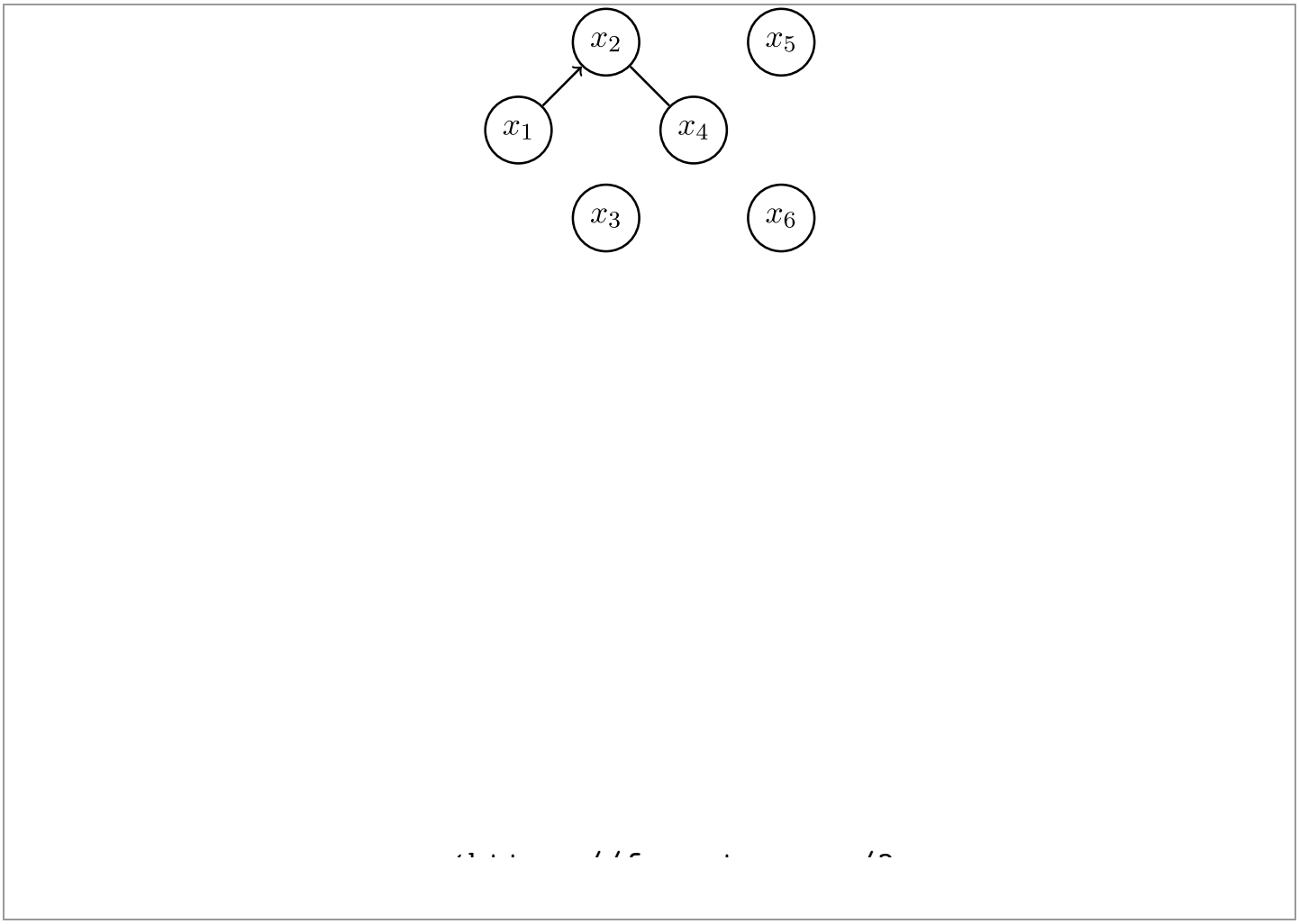
We can also notice that the lines appear to be too thin and that it's difficult to distinguish between directed and undirected edges. We can make the graph more readable by passing the parameter *thick* to the *tikzpicture* environment, upon its introduction:

```
\begin{tikzpicture}[thick, main/.style = {draw, circle}]
...
\end{tikzpicture}
```

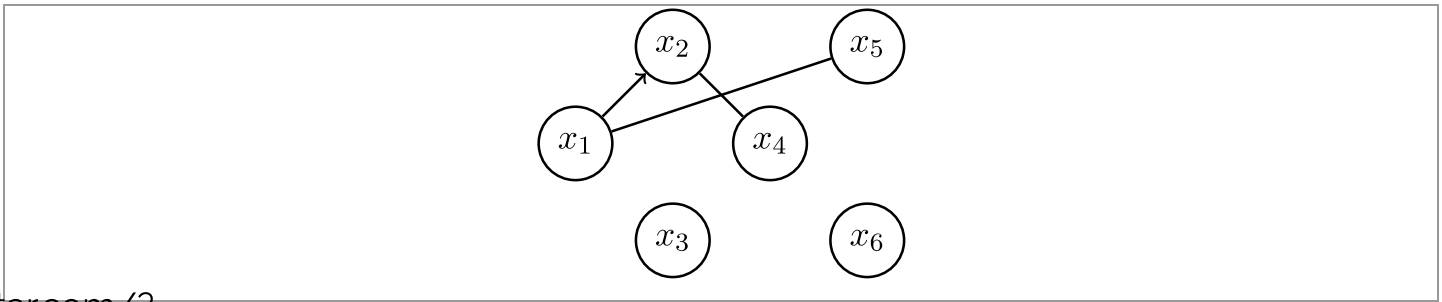
If the vertices appear too close to one another, we can also specify the parameter *node distance* to the *tikzpicture* environment to spread them further:

```
\begin{tikzpicture}[node distance={15mm}, thick, main/.style = {draw, circle}]
...
\end{tikzpicture}
```



3.4. More Challenging Edges

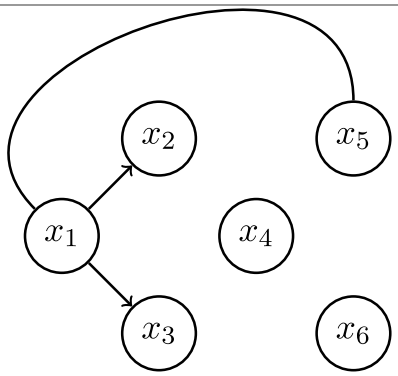
We may also want to connect with an edge some nodes for which the straight path intersects other edges or nodes. Say we try to connect with a straight path the nodes (1) and (5):



It doesn't look good. Instead, we want to loop around the graph, from the outside and above node 2, and then descend back towards the graph. Luckily, we can do that by specifying a parameter *looseness* after declaring the initial point of a line. To better

clarify where we want the edge to go, we also specify the angles *out* and *in* which indicate, respectively, the angle of the outgoing and ingoing edges (/cs/graph-theory-intro#5-endpoints-directions-loops):

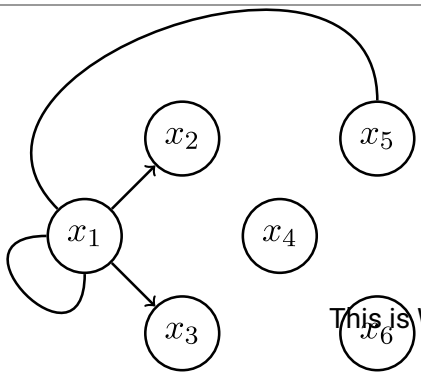
```
\draw (1) to [out=135,in=90,looseness=1.5] (5);
```



3.5. Loops

We can use the same technique to draw loops (/cs/graph-theory-intro#5-endpoints-directions-loops) in the graph, by indicating twice the same node as the starting and ending points of a loose line:

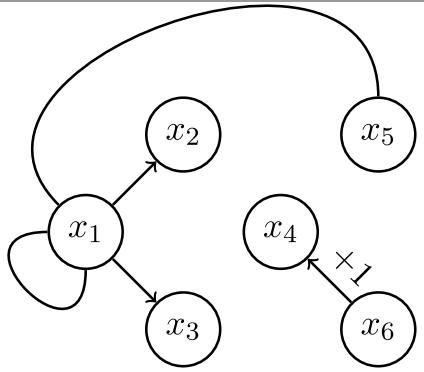
```
\draw (1) to [out=180,in=270,looseness=5] (1);
```



3.6. Draw Weighted Edges

If our graph is a weighted graph, we can add weighted edges as phantom nodes inside the *\draw* command:


```
\draw[->] (6) -- node[midway, above right, sloped, pos=1] {+1} (4);
```

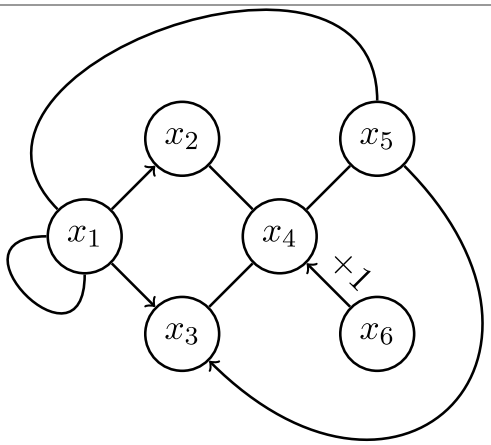


With this command, we're adding an invisible node halfway between vertices (6) and (4), and then to the top right of the edge's middle point. The label {+1} indicates the weight of the graph. The parameters *sloped* and *pos* indicate, respectively, that we want the weight to be orthogonal to the edge, and shifted by a certain amount along the edge. We can determine by trial and error what is the correct amount, according to the size of our image.

3.7. The Full Graph

And finally, we can repeat all the instructions above to populate the graph with all edges we need:

```
\documentclass{article}
\usepackage{tikz}
\begin{document}
\begin{tikzpicture}[node distance={15mm}, thick, main/.style = {draw, circle}]
\node[main] (1) {$x_1$};
\node[main] (2) [above right of=1] {$x_2$};
\node[main] (3) [below right of=1] {$x_3$};
\node[main] (4) [above right of=3] {$x_4$};
\node[main] (5) [above right of=4] {$x_5$};
\node[main] (6) [below right of=4] {$x_6$};
\draw[->] (1) -- (2);
\draw[->] (1) -- (3);
\draw (1) to [out=135,in=90,looseness=1.5] (5);
\draw (1) to [out=180,in=270,looseness=5] (1);
\draw (2) -- (4);
\draw (3) -- (4);
\draw (5) -- (4);
\draw[->] (5) to [out=315, in=315, looseness=2.5] (3);
\draw[->] (6) -- node[midway, above right, sloped, pos=1] {+1} (4);
\end{tikzpicture}
\end{document}
```



4. Neuralnetwork

Another useful package for generating graphs is Neuralnetwork (https://github.com/battlesnake/neural). This package, as the name ˙ ¸gests, is incredibly helpful to draw the architecture of a neural network (/cs/neural-net-advantages-disadvantages), and in particular of feedforward neural networks (/cs/ml-linear-activation-functions#structure-of-feed-forward-neural-networks).

The package simplifies the construction of layers and their manipulation. **It does so by shortening all commands for a single layer to one line**.

Its main advantage is in the rapidity of describing a full graph with only a few lines of code. Instead, its main disadvantage lies in the scarce customizability of the individual characteristics of the graph.

4.1. Basics of the Package

In the LaTeX package Neuralnetwork, we define graphs inside an environment with the same name:

```
\begin{neuralnetwork}
...
\end{neuralnetwork}
```

Inside this environment, we can then add layers by using the *\inputlayer*, *\hiddenlayer*, and *\outputlayer* commands, according respectively to whether we refer to the first, hidden (/cs/neural-networks-hidden-layers-criteria), or last layer of a neural network. The syntax is similar for all three commands:

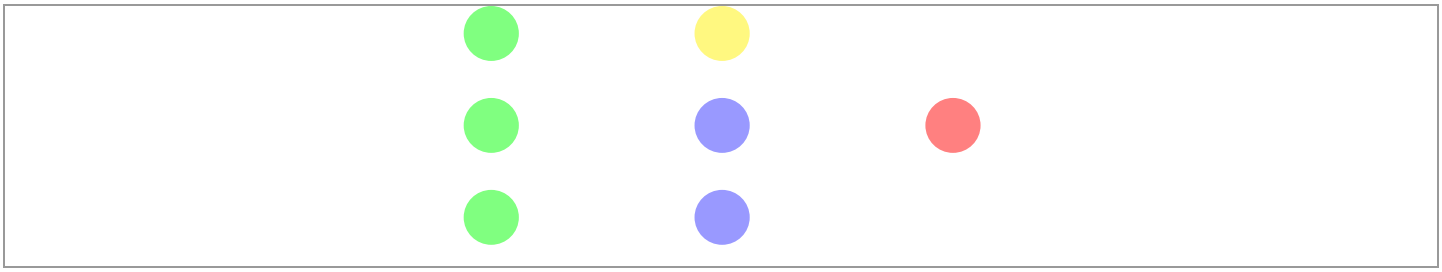

```
\inputlayer[count=howManyNodes, bias=trueOrFalse, nodeclass={classOfNode}] {}
```

The parameter *count* indicates the number of nodes in the given lay : ıe term bias indicates instead whether the bias (/cs/neural-networks-bias) for that ıayer should be counted as a separate element. In that case, this bias wouldn't be connected to the previous layer, when links are established. Lastly, the class of the node consists of either one of the predefined classes *input node*, *hidden node*, or *output node*, or also any of our custom classes.

4.2. Drawing a Neural Network

We can draw a simple neural network comprising of three features in the input, a hidden layer with two neurons and a bias term, and a single neuron in the output layer. This resembles closely the structure of a neural network for XOR classification (/cs/neural-networks-hidden-layers-criteria#4-neural-networks-for-non-linear-separation):

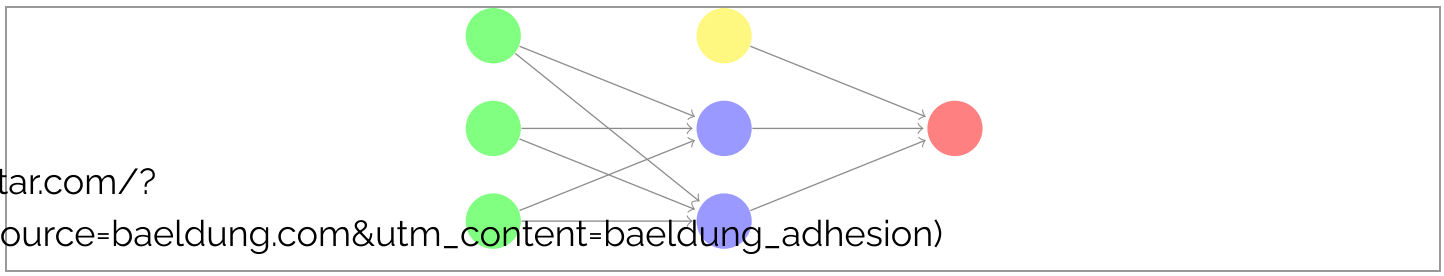
```
\documentclass{article}
\usepackage{neuralnetwork}
\begin{document}
\begin{neuralnetwork}
\inputlayer[count=3, bias=false]{}
\hiddenlayer[count=2]{}
\outputlayer[count=1]{}
\end{neuralnetwork}
\end{document}
```



4.3. Adding Edges and Labels

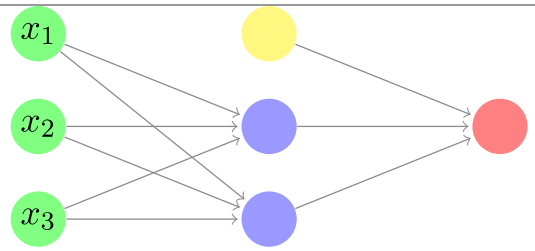
We can also add edges between each layer, by following each layer that we want to connect with its preceding layer by the *\linklayers* command:

```
...
\inputlayer[count=3, bias=false]{} \linklayers
\hiddenlayer[count=2]{} \linklayers
\outputlayer[count=1]{}
...
```



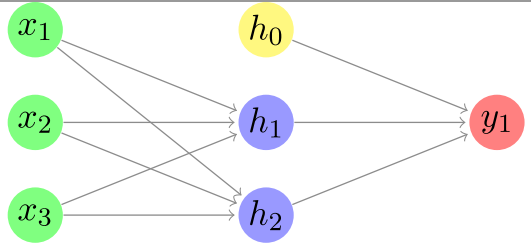
Notice how the yellow node, indicating the bias term in the intermediate layer, isn't connected to the input layer. We can also add labels to the nodes, though this is somewhat tricky. The way to do it is by defining a new LaTeX command, and passing it as a parameter *text* to a layer:

```
...
\newcommand{\x}[2]{$x_#2$}
\inputlayer[count=3, bias=false, text=\x]
...
```



We can also follow the same criterion with the hidden layer and the output layer by creating appropriate commands:


```
...
  \newcommand{\h}[2]{$h_#2$}
  \newcommand{\y}[2]{$y_#2$}
...
  \hiddenlayer[count=2, text=\h]{} \linklayers
  \outputlayer[count=1, text=\y]{} \linklayers
...
```

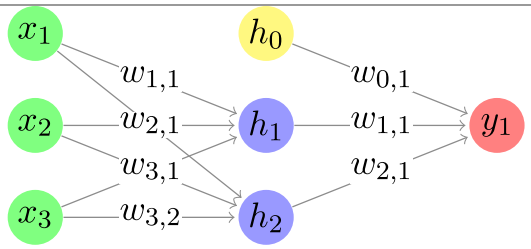


The bias term in the hidden layer automatically takes the index of h_0, as is common in the literature, without having to specify any particular instructions.

4.4. Weights

We can also add weights to the edges by using a formula like the one below. We have to define a new command first, and then assign it to the default text for links:

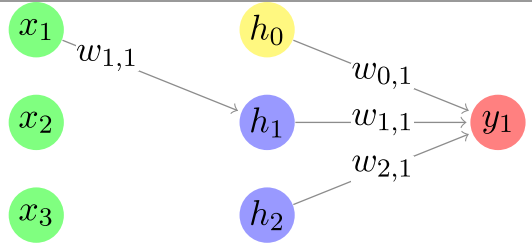
```
\newcommand{\w}[4] {$w_{#2,#4}$}
\setdefaultlinklabel{\w}
```



Here, #1 indicates the layer of origin, #2 the node of origin, and #3 and #4 indicate layer and node of destination, respectively. Notice that, while the weights of the hidden layer look normal, some weights in the previous layer are however missing.

This is because they're automatically rewritten by some other weights, and as a consequence, we only see some of them. The only way to address this problem is by defining all links individually and then giving them the appropriate label. We can do this by using the *\link* command, instead of *\linklayers*.


```
...
\inputlayer[count=3,bias=false, text=\x]{}
\hiddenlayer[count=2, text=\h]{}
\link[style={}, labelpos=near start, from layer=0, from node=1, to layer=1, to
node=1]
...
```

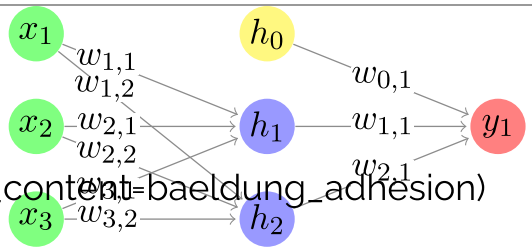


The parameter *labelpos* is the one we use to shift towards the node of origin. In this case, the label is assumed by the edge. This, in turn, avoids the overlapping between the edges' labels.

4.5. Finishing Touches

We can then repeat this operation with two nested *\foreach* loops, which will let us place all labels in the correct, readable positions:

```
\foreach \n in {1,...,3}{
    \foreach \m in {1,2}{
        \link[style={}, labelpos=near start, from layer=0, from node=\n, to
layer=1, to node=\m]
    }
}
```

The network is now complete. We can replicate it in a LaTeX editor with this full code:

```
\documentclass{article}
\usepackage{neuralnetwork}
\begin{document}
\begin{neuralnetwork}
\newcommand{\x}[2]{$x_#2$}
\newcommand{\y}[2]{$y_#2$}
\newcommand{\h}[2]{$h_#2$}
\newcommand{\w}[4] {$w_{#2,#4}$}
\setdefaultlinklabel{\w}
\inputlayer[count=3, bias=false, text=\x]
\hiddenlayer[count=2, text=\h]
\foreach \n in {1,...,3}{
    \foreach \m in {1,2}{
        \link[style={}, labelpos=near start, from layer=0, from node=\n, to
layer=1, to node=\m]
    }
}
\outputlayer[count=1, text=\y] \linklayers
\end{neuralnetwork}
\end{document}
```

5. Converting to LaTeX Through Extensions

There's also another option to mention that relates not so much to LaTeX packages themselves but, instead, to the automatic conversion into LaTeX code of graphs created in other graphical editors. This is particularly convenient if we're used to drawing the graphs in other programming languages, and we're still learning how to draw graphs in LaTeX.

There are, in fact, libraries or extension for the automatic conversion into LaTeX code of the graphical output of another editor:

- In Python, we can convert graphs created with ***matplotlib (https://matplotlib.org/)*** into LaTeX, thanks to the library ***tikzplotlib (https://github.com/nschloe/tikzplotlib)***
- In MatLab, we can use the scripts from ***matlab2tikz (https://us.mathworks.com/matlabcentral/fileexchange/22022-matlab2tikz-matlab2tikz)*** to perform a similar operation
- In Blender 2.4, we can use ***blend2tikz (https://github.com/xyz2tex/blend2tikz)*** to export Blender curves into LaTeX code

6. Conclusion

In this article, we studied how to draw graphs using LaTeX.

2 COMMENTS

Oldest

View Comments

Comments are closed on this article!

CATEGORIES

ALGORITHMS (/CS/CATEGORY/ALGORITHMS)
ARTIFICIAL INTELLIGENCE (/CS/CATEGORY/AI)
CORE CONCEPTS (/CS/CATEGORY/CORE-CONCEPTS)
DATA STRUCTURES (/CS/CATEGORY/DATA-STRUCTURES)
GRAPH THEORY (/CS/CATEGORY/GRAPH-THEORY)
LATEX (/CS/CATEGORY/LATEX)

NETWORKING (/CS/CATEGORY/NETWORKING)
SECURITY (/CS/CATEGORY/SECURITY)

SERIES

ABOUT

ABOUT BAELDUNG (HTTPS://WWW.BAELDUNG.COM/ABOUT)
THE FULL ARCHIVE (/CS/FULL_ARCHIVE)
EDITORS (HTTPS://WWW.BAELDUNG.COM/EDITORS)

TERMS OF SERVICE (HTTPS://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
PRIVACY POLICY (HTTPS://WWW.BAELDUNG.COM/PRIVACY-POLICY)
COMPANY INFO (HTTPS://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)