An introduction to automata design with TikZ’s automata library

Igor Borja

Abstract
This article is a quick introduction to TikZ’s automata library, used for the design and typesetting of finite automata in LaTeX. It also explores the use of LaTeX loops and conditionals to automate the generation of images that follow noticeable patterns.

TikZ itself is a package used for generating a variety of figures—from geometry configurations to graphs and automata—allowing for more control over image editing and quality. Although the package is very versatile, its uses for designing automata will be the primary topic of this article.

1 Introduction and basic syntax
Finite automata, also called finite state machines, are a basic concept in computer science for modeling computation. Wikipedia (en.wikipedia.org/wiki/Finite-state_machine) provides an introduction to the topic.

In this article, all the code to typeset an automaton will be contained inside a \begin{tikzpicture} environment [3]. After starting the environment, you can pass optional arguments, separated by a comma, such as node distance and arrow style. A reminder: the node distance (which we’ll see below) must be a dimension (cm, em, pt, etc.).

1.1 Nodes
You can declare a node of an automaton via the following syntax:

\node[state, \langle state modifiers \rangle, \langle position modifiers \rangle] \{\langle id \rangle\} \{\langle name \rangle\};

Note that a node declaration should always end in a semicolon. Let’s analyse all of these parameters:

1. Every state node must begin with the word state.
2. State modifiers are mostly used to indicate that node is the initial node or an accepting node, and thus are often not needed.
3. Position modifiers are used to place a node relative to another (already declared) node. Some common modifiers are right=of \langle id \rangle, left=of \langle id \rangle, below=of \langle id \rangle and above=of \langle id \rangle. Here, \langle id \rangle is the id of the node relative to which the positioning is carried out.

Also, it’s possible to give dimensions via the xshift and yshift parameters to achieve manual control over the position after the relative placement.

1.2 Edges
You can declare an edge between two nodes with the following syntax

\draw \langle id-head \rangle edge[\langle options \rangle] node[\langle options \rangle] {\langle value \rangle} \langle id-tail \rangle;

Every sequence of consecutive edge declarations must be preceded by a \draw command. Also, a semicolon is used to indicate the last edge declaration of a sequence; any that come after that and before another \draw command will be ignored. Therefore (provided \text{n1, n2, n3} have all been declared), this is correct:

\draw (n1) edge node{text} (n2)
(n2) edge node{more text} (n3)
(n2) edge node{more text} (n1);
\node[state, above=of n1] \{n4\} \{text\};
\draw (n3) edge node{more text} (n4);

While this is (for both reasons mentioned above) not correct:

4. The \langle id \rangle is the unique identifier that will be used to refer to that node later.
5. The \langle name \rangle is the text that will appear in the automaton, inside the circle that represents that node. It does not need to be plain text; it’s also possible to use a math expression (enclosed by single $ signs).

A minimal working example:
\documentclass{standalone}
\usepackage{tikz}
\usetikzlibrary{automata}
\usetikzlibrary{positioning, arrows}
\begin{tikzpicture}
\[->, node distance = 2cm\]
\node[state] \{p = 1\};
\node[state, accepting, right=of p, yshift= -2cm] \{q = 2\};
\end{tikzpicture}

\begin{figure}[h]
\centering
\begin{tikzpicture}
\node[state, above=of n1] \{text\};
\end{tikzpicture}
\caption{Two nodes}
\end{figure}

The use of \langle direction \rangle of=\langle id \rangle, although correct, is marked as deprecated in PGF/TikZ source code [4].

Igor Borja
doi.org/10.47397/tb/44-1/tb136prado-automata
1.2.1 Analysing the different components of an edge command

\(\langle \text{id-head} \rangle\) is the identifier of the head node — the node from which the edge is originated.

\(\langle \text{id-tail} \rangle\) is the identifier of the tail node — the node to which the edge arrives.

The options after the edge keyword indicate how the edge should be drawn. The following options are the most common:

- **Directions** right, left, below, above: indicate from where the edge should leave
- **loop**: specifies that the edge should loop and go back to the head node. Using the loop option makes Ti\(\text{kZ}\) ignore the tail node’s id (which can be left empty).

Combining the loop option with directions indicates where the loop should be rendered:
- above, to the right, left or below the node.
- **bend**: indicates bends in the edge to a certain direction

The options after the node keyword indicate how the text associated with that edge should be drawn. The most common arguments are the four main directions right, left, above and below.

Defaults: if no edge-positioning options are given, the edge will be drawn as a straight line by default. Also, if no node options (i.e., options that determine the positioning of the edge text) are provided, the text will be rendered at the “center” of the edge by default. See both behaviors below:

\begin{tikzpicture}
\[->, node distance = 2cm\]
\node[state, above=of p] (p) {$p = 1$};
\node[state, accepting, right=of p, yshift=-2cm] (q) {$q = 2$};
\draw
(p) edge node{hello} (q);
(p) edge[loop] (p);
\end{tikzpicture}

Figure 2: Two nodes, a labeled edge and a loop

2 Loops and automation by example

Here we show a more complex example.

2.1 Context

Consider the language \(L\) over the alphabet \(\Sigma = \{a, b\}\) that contains all the sequences with at least 2 characters \(a\) and at least 1 character \(b\). Note that \(L\) is the intersection of two regular languages (over the same alphabet \(\{a, b\}\)): the set of strings with at least 2 characters \(a\) and the set of strings with at least 1 character \(b\).

Therefore, using the construction detailed in \[2\], a possible finite automaton that recognizes \(L\) is \(M = (Q, \{a, b\}, \delta, q_0, 0, q_2, 1)\), where

\[Q = \{q_{i,j} \mid 0 \leq i \leq 2 \wedge 0 \leq j \leq 1\}\]

are the states that can be reached. The transition function \(\delta\) works as follows:

\[\delta(q_{i,j}, a) = q_{\min(2,i+1),j}\]

and

\[\delta(q_{i,j}, b) = q_{i,\min(1,j+1)}\]

In other words, \(q_{i,j}\) represents that the string read has (up to that point) \(i\) characters \(a\) (if \(i < 2\) or 2 or more, if \(i = 2\)). Also, it has \(j\) characters \(b\) if \(j < 1\), else it has 1 or more. Representing it graphically, we get the following state diagram:

Figure 3: State diagram

An introduction to automata design with Ti\(\text{kZ}\)’s automata library
which we can produce with the following \LaTeX{} code:

\begin{verbatim}
\begin{tikzpicture}
[-, node distance = 1.3cm]
\node[state, initial] (a0b0) {$q_{0,0}$};
\node[state, right=of a0b0] (a1b0) {$q_{1,0}$};
\node[state, right=of a1b0] (a2b0) {$q_{2,0}$};
\node[state, above=of a0b0] (a0b1) {$q_{0,1}$};
\node[state, above=of a1b0] (a1b1) {$q_{1,1}$};
\node[state, above=of a2b0] (a2b1) {$q_{2,1}$};
\node[state, accepting, above=of a2b0] (a0b2) {$q_{0,2}$};
\node[state, accepting, above=of a1b0] (a1b2) {$q_{1,2}$};
\node[state, accepting, above=of a2b1] (a2b2) {$q_{2,2}$};

%% Horizontal edges in first layer
\draw
(a0b0) edge node[a] {} (a1b0);
(a1b0) edge node[a] {} (a2b0);
(a0b0) edge node[b] {} (a0b1);
(a1b0) edge node[b] {} (a1b1);
(a2b0) edge node[b] {} (a2b1);

%% Vertical edges
\draw
(a0b0) edge node[above] {} (a0b1);
(a1b0) edge node[above] {} (a1b1);
(a2b0) edge node[above] {} (a2b1);
\node[state, initial] (a0b0);
\node[state, accepting, above=of a2b0] (a0b2);
\node[state, accepting, above=of a1b0] (a1b2);
\node[state, accepting, above=of a2b1] (a2b2);
\end{tikzpicture}
\end{verbatim}

It is easy to see that this state diagram’s grid-like structure generalizes nicely to the family of languages $\{L_{m,n}\}_{m,n \in \mathbb{N}}$, where $L_{m,n}$ is the language of strings with at least $m$ characters $a$ and at least $n$ characters $b$.

However, drawing the state diagram for $L_{m,n}$ quickly becomes too much work, since there will be $(m+1)(n+1)$ nodes and $2(m+1)(n+1)$ edges: even at small values (say, $m = 4$ and $n = 3$) it is still more than 60 lines of code.

### 2.2 Loops

However, due to its very simple structure, rendering automata like this one can be abstracted and automated in a relatively straightforward way, using \texttt{foreach} loops, available through the package \texttt{pgffor}.

1. For the nodes, it’s possible to draw first $q_{0,0}$ with id $a0\sim b0$, then for each $1 \leq i \leq m$ draw $q_{i,0}$ at the right of $q_{i-1,0}$ and attribute to it the id $a\langle i\rangle b0$ (where $i$ is a placeholder for the value of $i$). This completes the first row.

Then, for each $1 \leq j \leq n$ and for each $0 \leq i \leq m$ we draw $q_{i,j}$ above $q_{i,j-1}$ and attribute to it the id $a\langle i\rangle b\langle j\rangle$. At each iteration it is necessary to check if $i = m$ and $j = n$, in which case we need to add the \texttt{accepting} option.

2. For the edges, we have four different cases: standard horizontal edges, standard vertical edges, edges that loop above the node and edges that loop at the right of the node.

For each $0 \leq i \leq m - 1$ and $0 \leq j \leq n$, we build a horizontal edge from $q_{i,j}$ to $q_{i+1,j}$. Then, for each $0 \leq i \leq m$ and $0 \leq j \leq n - 1$ we build a vertical edge from $q_{i,j}$ to $q_{i,j+1}$. Finally, we build an edge that loops above $q_{i,n}$ for each $0 \leq i \leq m$ and an edge that loops at the right of $q_{m,j}$ for each $0 \leq j \leq n$.

It is necessary to refer to $i - 1$ and $j - 1$ several times in this algorithm. However, simply using the \TeX{} code $i - 1$ and $j - 1$ and $a\backslash i - 1 b\backslash j - 1$ (for ids) in the \TeX{} code won’t work: loop indices in \texttt{foreach} loops function as standard variables defined with \texttt{\def} (just with restrained scope). That means that any reference \texttt{\i} will be replaced by the value of \texttt{\i} (as a string), so the expression $\backslash i - 1$ will not be evaluated. For example, if $i = 5$, $\backslash i - 1$ will be replaced by $5 - 1$.

In order to fix that, we use the fixed-point arithmetic package called \texttt{fp} and its command for evaluating expressions: \texttt{\fp_eval}. The implementation below summarizes all that in working \LaTeX{} code, abstracting it all in a command with three arguments called \texttt{\gridAutomata}. The first argument is the node distance, the second is $m$ and the third is $n$:

\begin{verbatim}
\usepackage[ifthen]
newcommand[\gridAutomata][3]{2cm}
{\begin{tikzpicture} [ ->, node distance = \1, ]
% building first row of nodes
ifthenelse{equal[\2]{0}}{
  \node[state, initial, ...
end{verbatim}
%% Rightmost loops
\foreach \j in {0,...,#3}
{
  \draw (a#2-b\j) edge[loop right]
  node[right]{a} (a#2-b\j);
}

%% Horizontal edges
\foreach \i in {0,...,fpeval{#2 - 1}}
{
  \foreach \j in {0,...,#3}
  {
    \draw (a\i-b\j) edge
    node[above]{a} (a\i-b\fpeval{\i+1}-b\j);
  }
}

%% Vertical edges
\foreach \j in {0,...,fpeval{#3 - 1}}
{
  \foreach \i in {0,...,#2}
  {
    \draw (a\i-b\j) edge
    node[right]{b} (a\i-b\fpeval{\j + 1});
  }
}
\end{tikzpicture}

Using this command for \(m = 4, n = 3\) with a node distance of 1.5 cm yields the result below:

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{automaton-m4n3}
\caption{Result for \(m = 4, n = 3\)}
\end{figure}

This command’s generality allows for construction of a larger example. The image on the next page uses the values \(m = 7, n = 8\) with a node distance set to 1.5 cm.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{automaton-m7n8}
\caption{Result for \(m = 7, n = 8\)}
\end{figure}

An introduction to automata design with Ti\kz’s automata library
Figure 5: Result for $m = 7, n = 8$
3 More customization possibilities

3.1 Absolute positioning

It is also possible to define the position of each node manually, as a coordinate pair. The syntax is:
\node[state] at (⟨x⟩, ⟨y⟩) {⟨id⟩} {⟨name⟩};

The fields ⟨x⟩ and ⟨y⟩ are the x and y components of the position. To better illustrate this, we make use of the help lines option to draw a 3 × 3 grid, in which the nodes will be positioned.

Figure 6: An isosceles triangle

\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,3);
\node [state] at (0, 0) (a) {$a$};
\node [state] at (1, 2) (b) {$b$};
\node [state] at (3, 1) (c) {$c$};
\draw
(a) edge (b)
(b) edge (c)
(c) edge (a);
\end{tikzpicture}

3.2 Drawing arcs and bending edges

Especially when avoiding crossing edges in more complicated automata, it is useful to be able to draw edges as arcs (instead of straight lines). In plain TiKZ it is possible to render arbitrary arcs, specifying the center, the starting and stopping angle, and the radius (as shown in [5]):
\draw ⟨(x),⟨y⟩⟩ arc ⟨start⟩:(stop):(radius);

Through the automata library, however, the syntax is simplified for the case of an arc between two (already defined) nodes.
\begin{tikzpicture}
\node [state] at (0, 0) (a) {$a$};
\node [state] at (1, 1) (b) {$b$};
\node [state] at (3, 1) (c) {$c$};
\draw
(a) edge[bend left=15] node[above left] {$15^\circ$ arc} (b)
(b) edge[bend left=90] node[above=0.25] {$90^\circ$ arc}(c)
(c) edge[bend left=60] node[below] {$60^\circ$ arc} (a);
\end{tikzpicture}

\begin{figure}[h]
\centering
\begin{tikzpicture}
\node[state] at (0, 0) (a) {$a$};
\node[state] at (1, 2) (b) {$b$};
\node[state] at (3, 1) (c) {$c$};
\draw
(a) edge[bend left=15] node[above left] {$15^\circ$ arc} (b)
(b) edge[bend left=90] node[above=0.25] {$90^\circ$ arc}(c)
(c) edge[bend left=60] node[below] {$60^\circ$ arc} (a);
\end{tikzpicture}
\caption{Various arcs}
\end{figure}

References


\diamond Igor Borja
igorpradoborja (at) gmail dot com
https://github.com/IgorPBorja