Typesetting figures for computer science
Andrew Mertz, William Slough and Nancy Van Cleave

Abstract
Presentation of concepts from computer science can benefit from informative diagrams and figures. These include trees, graphs, logic circuits, stacks and stack frames, algorithms expressed with pseudocode, code listings and memory layout, for example. Producing these types of diagrams can sometimes be challenging. Fortunately, there are a number of \LaTeX packages that can be used for this purpose. In this paper, we will illustrate the use of a few packages — including \texttt{tikz}, \texttt{bytefield}, \texttt{forest}, \texttt{drawstack}, and \texttt{listings} — that are well-suited for constructing high-quality figures for computer science.

1 Introduction
Popular wisdom dictates that \textit{a picture is worth a thousand words}. Illustrations can be an indispensable aid to understanding new concepts, and never more so than in the classroom. The types of figures we most often use in our computer science courses fall mainly into one of four categories:

- systems: logic circuits, instruction set details, and stack frames
- algorithms/data structures: pseudocode, graphs, binary trees, and search trees
- theory of computation: automata, grammars, and parse trees
- discrete mathematics: graphs and trees

In all courses, there is a need for clearly presenting algorithms and code.

There are two approaches one can take: either generate a graphic with some third-party software, export it in an appropriate format and subsequently include it in the \LaTeX source with the \texttt{graphicx} package [1], or generate the graphic with one of the “friends” of \TeX, such as \texttt{METAPost} [6] or \texttt{Ti\kZ} [16]. Where practical, we prefer the latter approach, since the results blend well with the surrounding typeset text and produce vector graphics, contributing to their quality. This quality derives from the fact that images generated in this manner use the same fonts as the typeset text and can utilize mathematical typesetting as needed.

Additionally, there are a number of packages designed for special purposes — such as typesetting trees, stacks, and graphs, to name a few. Some of the more recent packages on CTAN make use of \texttt{TikZ}, adding a syntax layer to make them easier to use while preserving excellent typeset results.

The number of packages available from CTAN can be somewhat overwhelming to a \LaTeX user with a specific task in mind. In a recent search, we found 35 packages for the topic \textit{tree}. How does one choose from this embarrassment of riches? While references such as [4] and [10] certainly provide useful guidance they are snapshots in time, so a number of packages do not appear there.

Our purpose here is to provide examples of a number of typesetting tasks that are of interest to computer scientists. For each such task, we have identified packages we feel are especially appropriate. Although we are not providing tutorial introductions to these packages, we hope these examples may introduce readers to a few unfamiliar packages and provide motivation for further exploration.

2 Typesetting code
We often have a need to typeset code in a specific programming language, such as Python or Java. For such situations, we recommend the use of the \texttt{listings} package [5].

The \texttt{listings} package, introduced in 1996, is relatively mature and still enjoys current support. While perhaps best-known for its ability to produce “pretty-printed” output, verbatim-like results can also be produced. It provides support for more than 100 programming languages and dialects, including \LaTeX, with the ability to define styles for yet others.

It is easy to get started with this package, knowing just two commands and one environment:

\begin{verbatim}
\lstset
\lstinputlisting
{lstlisting}
\end{verbatim}

Appearance of typeset code is controlled with \texttt{\lstset}, which provides options for languages, colors, font sizes and styles, line numbering, and a host of other possibilities. Desired options are specified using a comma-separated list of key/value pairs, as follows:

\begin{verbatim}
\lstset{key1=value1,...}
\end{verbatim}

An example of this is shown in Figure 1, where a number of such options are specified: \texttt{language}, \texttt{showstringspaces}, \texttt{columns}, etc. To typeset code, we can use either \texttt{\lstinputlisting} or \texttt{lstlisting}, the difference being where the code to be typeset is located. The command

\begin{verbatim}
\lstinputlisting{filename}
\end{verbatim}

typesets the contents of the file \texttt{filename}, using the options previously requested. In contrast, code appears directly within an \texttt{lstlisting} environment:

\begin{verbatim}
Typesetting figures for computer science
\end{verbatim}
def gcd(p, q):
    """
    Computes the greatest common divisor of
two nonnegative integers p and q using
Euclid’s method.
    """
    if (q == 0):
        return p
    else:
        remainder = p % q
        return gcd(q, remainder)

Figure 1: Typeset Python code using the listings package, producing pretty-printed output.

\begin{lstlisting}
code to typeset
\end{lstlisting}

Revisiting Figure 1, we see how Euclid’s algorithm for computing the greatest common divisor, as implemented in Python and stored in the file named gcd.py, can be typeset.

Most of the options specified here are evident from the typeset result: a single frame enclosing the code, tiny line numbers on the left, etc. Two of these options, however, are perhaps less obvious—showstringspaces and columns. Setting the first of these to false prevents the visible space (\ ) from appearing within any of the strings in the code. To understand the columns setting, it helps to know that there are two broad categories possible: fixed and flexible. A fixed column format adjusts the space between letters in an attempt to align columns; a flexible format makes no such attempt. Of the flexible formats available, we prefer fullflexible.

As a second example of the listings package, refer to Figure 2. Here, we did not specify a language, so no keywords are highlighted. To obtain a verbatim-like appearance, basicstyle is set to a monospace font. Setting the keepsparse option causes spaces which appear in the code to be respected which, in turn, preserves column alignment and indenting.

The listings package has many other options and advanced capabilities. For example, it is possible to include TeX markup within a listing by providing “escaped” code, making it possible to blend mathematical comments with code.

3 Typesetting algorithms

To display algorithms in pseudocode, we think the style exhibited in [3] (famously known as CLRS, after the authors’ initials) is quite attractive, recognizing that its use requires some markup effort. For this, the clrscode3e package [2] can be used. An older version, clrscode, is also available; the 3e version matches the style used by the 3rd edition of CLRS.

Presenting an algorithm in this style is done using a codebox environment, as follows:

\begin{codebox}
algorithm, with markup
\end{codebox}
\Procname{$\proc{Euclid-gcd}(p, q)$}
\zi \Comment Pre: $p$ and $q$ are two nonnegative integers.
\zi \Comment Post: $\proc{Euclid-gcd}(p, q) = \func{gcd}(p, q)$.
\li \If $q \isequal 0$
\li \Then
\li \Return $p$
\li \Else
\li \gets p \bmod q$
\li \Return $\proc{Euclid-gcd}(q, \id{remainder})$
\End

\begin{figure}[h]
\centering
\begin{verbatim}
\Euclid-gcd(p, q)
  // Pre: $p$ and $q$ are two nonnegative integers.
  // Post: $\Euclid-gcd(p, q) = \gcd(p, q)$.
  \if $q == 0$
    \return $p$
  \else
    \remainder = $p \mod q$
    \return $\Euclid-gcd(q, \remainder)$
\end
\end{verbatim}
\caption{An algorithm presented with clrscode3e.}
\end{figure}

Figure 3 shows sample markup for presenting an algorithm with this package; the typeset result is in Figure 4. A few remarks will clarify some of the markup details being used here. Consult the documentation provided with this package for more complete information.

$\Euclid-gcd$ is the name of this algorithm, so it is being identified as such with the $\proc$ (procedure) macro. In a similar way, $\id{remainder}$ is an identifier indicated by $\id$. Each of $\li$ and $\zi$ commands cause new lines to begin, either numbered or not. The package also supplies miscellaneous commands such as $\gets$ and $\isequal$ which produce assignment and equality operators. Commands for control structures, such as the conditional shown here, mirror those found in modern programming languages.

4 Logic circuits

Circuits consisting of $\and$, $\or$, $\not$, and $\nand$ gates are discussed in our systems course and also play a minor role in one of our general education mathematics courses. We have recently been looking at ways to produce diagrams of these types of circuits.

$\TiKZ$ provides support for logic circuits and produces nice results. In this context, a circuit consists of a number of nodes and a collection of interconnections. The placement of the nodes can be specified either in absolute or relative coordinates. Using relative coordinates is handy, as circuits often consist of components arranged either horizontally or vertically. Since the interconnections can be specified symbolically (e.g., “the output of the first and gate is connected to the first input of the or gate”), it is easy to stretch or compress the final drawing with just a minor change.

To illustrate, we will dissect some of the code details required to draw a circuit which computes a 3-input majority function. In the discussion that follows, refer to Figures 5 and 6. Since we want to use the U.S.-style gates found in the circuits library of $\TiKZ$, the following is needed in the preamble:

\begin{verbatim}
\usetikzlibrary{circuits.logic.US}
\end{verbatim}

Figure 5: Some preliminary definitions for the majority circuit.
Figure 6: A logic circuit which computes the majority function: \( f(x, y, z) = \text{true} \) if two or more of its inputs are true; otherwise it is false.

indicates a two-input gate in which each input is “normal”; i.e., not inverted. The node style named \texttt{dot} will be used to produce a small, filled circle to indicate an electrical connection.

Moving on to Figure 6, let’s examine how this circuit is specified. The \texttt{tikzpicture} environment specifies \( x \) and \( y \) units of 1 cm each, which could easily be adjusted to obtain a stretched or compressed variant. The text height and depth which appears is present to ensure that the three inputs, \( x \), \( y \), and \( z \) are typeset on the same baseline.

Within the body of the environment, the comments provide some guideposts for understanding how the circuit is drawn. A few additional remarks may be helpful. Inputs \( x \), \( y \), and \( z \) are placed at absolute coordinates, spaced one centimeter apart horizontally. The three \texttt{and} gates are placed relative to the \( z \) input: “over 1, down 1”, “over 1, down 2”,...
and “over 1, down 3”. The or gate is placed in a similar way.

The interconnections are simply vertical and horizontal line segments, specified with the -- and |- line drawing capabilities of TiKZ: a single line segment is produced by --, whereas |- draws an ell-shape consisting of a vertical segment followed by a horizontal one.

To illustrate specific input values, we could use a \texttt{draw} option to easily add colors (e.g., red for \texttt{false}, blue for \texttt{true}) to this circuit. For example, replacing the \texttt{draw} commands with \texttt{draw[red]} will change the color of the lines and dots.

\section{Stacks and stack frames}

To understand function calls at the machine-language level, visualization of stacks and stack frames within memory is key. The package \texttt{drawstack} \cite{11} provides a means to illustrate memory with annotations and explicit links. Refer to Figure 7 for an example of a simple stack frame.

This package provides the \texttt{drawstack} environment. Each frame is enclosed within a pair of commands: \texttt{startframe} and \texttt{finishframe}. The argument provided to \texttt{finishframe} specifies the brace label. The brace itself groups all of the cells within a given frame. In this first example, just one frame appears, but any number of frames may be produced.

Each component of a frame consists of a \texttt{cell}, an optional \texttt{comment}, and an optional cell \texttt{pointer}. The comment and pointer appear to the cell’s right. Cell comments are useful for memory addresses or offsets, for example. In this example, we are using \texttt{tikzset} to specify an arrowhead of a customized size. Other features of TiKZ can be used to add additional arrows, change colors, and so on. Figure 8 provides a glimpse of the possibilities.

\subsection{Displaying fields of bits}

In computer systems courses students are introduced to machine language instructions, written in binary. These instructions are subdivided into fields of bits and given mnemonic names. Similarly, memory maps, network protocols, and file formats all consist of fields of data. The package \texttt{bytefield} \cite{12} can be used to show these layouts, and provides a wide variety of options to do so.

The \texttt{bytefield} environment may be used to create diagrams similar in format to Figure 9. The argument, 13 in this case, specifies the width of the figure in bits. The \texttt{bitwidth} option controls the amount of horizontal space given to each bit. Within this environment, \texttt{bitbox} is used to add a field one or more bits wide in a single row. Double slashes are used to end a row in a \texttt{bytefield} diagram, much like a \texttt{tabular} environment.

Figure 10 illustrates how \texttt{bytefield} can be used to format a machine instruction; in this example, from a hypothetical computer called LC-3 \cite{13}. The indices are added with \texttt{bitheader}, while the \texttt{endianness} option reverses the order of the indexing.

Groups can be used to add labels that span multiple rows, and can be placed on the left or the right of the diagram. Groups do not have to nest properly, unlike most environments. Use the \texttt{leftcurly}
Saved R1
Saved R0
PFP = x5f17
Return Address = x305E
N = 2
Saved R1
Saved R0
PFP = x5ffe
Return Address = x305E
N = 3
Saved R1
Saved R0
PFP = x0000
Return Address = x3011
N = 4

Figure 8: Linked stack frames corresponding to a recursive computation of FACT(4).

\begin{bytefield}[bitwidth = 1.2em, leftrightcurly = .]{16}
\bitheader[endianness = big]{0-15}\{
\bitbox{4}{0001}\ \\
\bitbox{4}{DR}\ \\
\bitbox{4}{SR1}\ \\
\bitbox{4}{0}\ \\
\bitbox{4}{00}\ \\
\bitbox{4}{SR2}\ \\
\end{bytefield}

ADD
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0001 DR SR1 00 SR2

Figure 10: Formatting a machine instruction using bytefield.

7 Automata
Central to the theory of computation is the concept of various types of automata, such as finite-state machines, pushdown machines, and Turing machines. These abstract computing devices are typically presented as directed graphs with labeled edges. These kinds of diagrams are good for understanding the static aspect of these machines.

How might we gain further appreciation of the dynamic aspect of automata—that is, the nature of how these machines process input strings? One way is to utilize a program like JFLAP [14], a Java-based
Typographic considerations for computer science education.

Formal language and automata package now often used in computer science education.

Using JFLAP, we can build automata using a convenient GUI interface, then explore their behavior on various input strings. In addition, we can experiment with constructions used in proofs, such as constructing an equivalent deterministic automaton given a non-deterministic one.

What options exist for incorporating a JFLAP-based automaton in a \LaTeX\ document? At present, the software exports in JPG, PNG, GIF, BMP and SVG formats. Unfortunately, these are less than ideal, as Figure 13 shows: fonts do not match those used in the document, “true” subscripts are not used, color may not be desired, and it is difficult to achieve accurate placement of the circular nodes, since JFLAP does not use a “snap to” grid for node layout.

Since TikZ has a library for drawing automata, we can imagine an export option from JFLAP which produces appropriate TikZ code which could then be included in a \LaTeX\ document. Such an option would be a vector-based format and allow for \LaTeX\-based markup, thereby eliminating most of the undesirable effects of using one of the bitmap formats now available. The possibility of inaccurately placed nodes would continue to be a problem.

Although not directly incorporated into the JFLAP software, the script jflap2tikz[8] achieves the goal of presenting automata created with JFLAP using TikZ-based graphics. The TikZ code output by jflap2tikz for our example automaton is shown in Figure 14, and Figure 15 shows the resulting processed output. The TikZ code produced by jflap2tikz is human-readable and thus can be further edited, if desired. The script allows for a “snap to” style grid which can improve alignment of the nodes. A large grid spacing corresponds to a coarser grid. Figure 16 shows the result of using such a grid on the example finite-state machine.

Figure 13: An example finite-state machine excerpted from[15], as exported by JFLAP using PNG format.
\begin{tikzpicture}
[>={Stealth[width = 6pt, length = 9pt]},
accepting/.style = {double distance = 2pt,
outer sep = 1pt + \pgflinewidth},
shorten >= 1pt,
auto]
\draw (62pt, -114pt) node[state, initial, initial text =](0){$q_{0}$};
\draw (163pt, -51pt) node[state](1){$q_{1}$};
\draw (184pt, -162pt) node[state](2){$q_{2}$};
\draw (275pt, -92pt) node[state, accepting](3){$q_{3}$};
\path[->] (0) edge[bend left] node{b}(2);
\path[->] (2) edge[bend left] node{a}(0);
\path[->] (1) edge node{a}(2);
\path[->] (0) edge node{$\lambda$}(1);
\path[->] (1) edge node{a}(3);
\end{tikzpicture}

Figure 14: Result of processing the source file for Figure 13 with the jflap2tikz script (slightly edited for space).

Figure 15: The finite-state machine from Figure 13, converted into TikZ and then processed as usual.

Figure 16: The finite-state machine from Figure 13, converted with a grid spacing of 100.

8 Trees

Forests and trees appear in a wide variety of forms in many areas of computer science. Drawing trees with GUI-style graphics software can be awkward and may yield imperfect results. Fortunately, there are a number of \TeX-based approaches. One of these involves the use of the forest package [17], which produces excellent results.

This package provides the forest environment, wherein a tree can be defined using bracket notation, a well-known syntax among linguists which captures the recursive nature of a tree. In this notation, a tree with a single root node \( r \) is represented by \([r]\). If a tree has more than a single node, then it has a root node \( r \) and \( n \) nonempty subtrees \( T_1, T_2, \ldots, T_n \). The bracket representation for this tree is \([r\ br(T_1) \ br(T_2) \ ldots \ br(T_n)]\), where \( br(T) \) is the bracket representation of tree \( T \).

Figure 17 illustrates how this package can be used to draw a tree. In this example, 6 is the root with two subtrees, \( T_1 \) and \( T_2 \), so the bracket notation for this tree is of the form \([6 \ br(T_1) \ br(T_2)]\). Using this same pattern to expand \( br(T_1) \) and \( br(T_2) \), we eventually obtain the bracketed form shown in the figure. Notice that spaces and newlines can be introduced to assist with readability and to help make clear the structure of the tree.

Another common structure in computer science is the binary tree, in which every node has at most two children, referred to as left and right children. The tree depicted in Figure 17 is not a binary tree, since it is not clear whether the leaf node 1 is a left child or a right child.

Indicating whether a node is a left or right child can be accomplished with the phantom option which reserves space for a node, but doesn’t draw an edge to it. Phantom nodes may occur anywhere in the tree: when one is used at the root it produces a forest. Figure 18 shows two phantom nodes, one as the left child of 8, the other as the right child of 7.
Trees are commonly drawn with circular nodes. This, too, is possible with the `forest` package since it is built upon TikZ. Thus, options from TikZ can be used to alter the result, and can be applied to individual nodes, a subtree, or the entire tree. To add circles to the tree of Figure 18, a first attempt would be to apply this option to the entire tree:

```latex
for tree = {draw, circle}
```

This would draw a circle at each node of the tree, but the size of each circle would depend on the dimensions of the text at each node—giving circles of varying sizes.

Specifying an appropriate minimum circle size, as shown in Figure 19, solves that problem, producing a tree with a uniform appearance.

It is also useful to be aware of the \texttt{fit} option. Figure 20 shows the same binary search tree formatted in two ways. On the left is the default result, also known as the \texttt{tight} fit, whereas the tree on the right shows the result with the \texttt{band} fit. Compact trees might be nice in many situations, but the wider tree on the right is the one typically encountered for search trees.

A red-black tree [3] is another type of binary tree, where the color of each node is important. Figure 21 displays a red-black tree. The text has been set to white, and the default color for nodes is black. The red nodes have the default color overridden with the option \texttt{fill = red} (printed in gray for this article).

9 Grammars and parse trees

Before looking at an example of a parse tree, we take a short detour into grammars. A context-free grammar is a formal system which describes a language as a set of rules.

The \texttt{syntax-mdw} package [18] may be used to typeset the syntax rules for a given language. A grammar environment is used, and produces nicely formatted output, as demonstrated in Figure 22. The \texttt{alt} command is used to specify alternative rules, and when typeset produces the | symbol, pronounced “or”. Setting the value of \texttt{grammarindent} determines the amount to indent each of the alternatives in the grammar definition.
for tree = {fit = band, circle, draw,  
          fill = black, text = white,  
          edge = {black, very thick}}

[51
 [21
  [19 [13, fill = red] [,phantom]]  
  [40 [24, fill = red] [,phantom]]
 ]
 [88
  [72, fill = red [61][86]]  
  [96 [90, fill = red] [,phantom]]
 ]

Figure 21: An example of a red-black tree.

\begin{grammar}
<expr> ::= <expr> '+' <expr>
| <expr> '-' <expr>
| <expr> '*' <expr>
| <expr> '/' <expr>
| '-'<expr>
| 'id'
\end{grammar}

\begin{forest}
[	extlangle expr\rangle
  [	extlangle expr\rangle
    [	extlangle expr\rangle
      [	extlangle expr\rangle
        ['id']
      ]
    ]
  ]
  [	extlangle expr\rangle
    ['id']
  ]
]
\end{forest}

Figure 22: The syntax rules of a simple grammar.

A parse tree represents the syntactic structure of a string according to some context-free grammar. Given the grammar in Figure 22, a parse tree can be used to represent an expression such as $(a + b) \times c$ as shown in Figure 23.

10 Combinatorial graphs

In graph theory and similar courses, combinatorial graphs must be produced in great numbers. The package tkz-graph [7] provides a variety of styles and macros for creating high-quality representations of graphs. The documentation is available only in French, but is so abundantly illustrated with well-done examples that it is accessible to those with little or even no knowledge of French.

Figure 24 displays several of the options available when drawing graphs. To select the vertex style (from a list of ten possibilities, for example Simple, Classic, or Shade), set \texttt{vstyle} to your choice using the \texttt{\GraphInit} command.

\% Expression: $(a+b)\times c$
\newcommand\E{$\langle expr \rangle$}
\newcommand\id{$\texttt{id}$}
\newcommand\plus{$\texttt{+}$}
\newcommand\mult{$\texttt{*}$}
\newcommand\lpar{$\texttt{(}$}
\newcommand\rpar{$\texttt{)}$}
\begin{forest}
\E
  [\E
    [\lpar
      [\E
        ['id']
      ]
    ]
    ['\plus']
    [\E
      ['id']
    ]
  ]
  ['\mult']
  [\E
    ['id']
  ]
\end{forest}

Figure 23: A parse tree for the expression $(a + b) \times c$, using the grammar from Figure 22.

Andrew Mertz, William Slough and Nancy Van Cleave
\begin{tikzpicture}
% Select vertex style
\GraphInit[vstyle = Classic]
% Indicate vertex color
\SetUpVertex[FillColor = gray!30]
% Set vertex size relative to label
\renewcommand*{\VertexInnerSep}{3pt}
% Position of vertex labels
\SetVertexLabelIn
% Establish Line width
\tikzset{EdgeStyle/.append style =
{line width = 2pt}}
% --- tkz-graph commands here ---
\end{tikzpicture}

Figure 24: Details of initializing values for a graph.

Alternately, macros such as \texttt{\VertexInnerSep} can be redefined. Also, several macros are provided to modify default options, for example, \texttt{\SetVertexLabelIn}. And finally, \texttt{\tikzset} can be used to modify both the vertex and edge styles.

Figure 25 illustrates the format of a simple graph using relative positioning to place the vertices. Vertex A is established, then all the other vertices are relative to A or the position of another vertex. It is of interest to note that the distance given to combined directions such as \texttt{\SOEA}, is applied in both directions. Thus \texttt{\SOEA[unit = 1](A){C}} results in vertex C being one unit south and one unit east of vertex A. Recall that this code resides in the \texttt{tikzpicture} environment following the code given in Figure 24. Hence, native TikZ commands such as

\texttt{\draw[help lines](0, 0) grid (4, -4);}

can be issued here.

An alternative way to place the vertices can be seen in Figure 26. Here the vertices are placed using absolute coordinates, and all edges can be included in the single \texttt{\Edges} macro.

By utilizing \texttt{\tikzset}, it is possible to design a new vertex style, as shown in Figure 27. Here the shape, size, and color have been redefined.

It is also possible to modify the vertex style within the \texttt{\SetVertexSimple} macro, as shown in Figure 28, where the shape, fill color, size and line have all been specified.

As Figure 29 shows, creating directed edges is as simple as setting the edge style. To improve the default arrowheads, \texttt{\tikzset} is used to select the \texttt{Stealth} shape, with the width and height enlarged to improve visibility. Since the edges are now directed, the order of vertices when creating edges becomes important.

Simple changes can affect the appearance of a graph. For example, changing the vertex style to \texttt{Shade} produces a sophisticated-looking graph as seen in Figure 30.

Typesetting figures for computer science
Figure 27: Designing your own vertex style, version 1.

\begin{tikzpicture}
\SetVertexSimple
\tikzset{VertexStyle/.style = {
    shape = rectangle,
    fill = gray,
    inner sep = 0pt,
    outer sep = 0pt,
    minimum size = 10pt}}
\Vertex(A) \EA(A){B}
\Edge(A)(B)
\end{tikzpicture}

Figure 28: Designing your own vertex style, version 2.

\begin{tikzpicture}[rotate = 18]
\SetVertexSimple[Shape = diamond,
FillColor = gray!25,
MinSize = 12pt,
LineWidth = 4pt,
LineColor = black!75]
\tikzset{VertexStyle/.append style = {
    inner sep = 0pt,
    outer sep = 2pt}}
\Vertices{circle}{A, B, C, D, E}
\Edges(A, B, C, D, E, A, C, E, B, D)
\end{tikzpicture}

Figure 29: Example of a directed graph.

Figure 30: A change in vertex style.

\begin{tikzpicture}[rotate = 18]
\SetVertexSimple[Shape = diamond,
FillColor = gray!25,
MinSize = 12pt,
LineWidth = 4pt,
LineColor = black!75]
\tikzset{VertexStyle/.append style = {
    inner sep = 0pt,
    outer sep = 2pt}}
\Vertices{circle}{A, B, C, D, E}
\Edges(A, B, C, D, E, A, C, E, B, D)
\end{tikzpicture}

% Set arrow type and size
\tikzset{>={Stealth[width = 3mm,
    length = 4mm]}}
% Set edge to arrow
\tikzset{EdgeStyle/.append style = {
    ->,line width = 1.5pt}}
% Nested squares of vertices as before
% Outer square - clockwise
\Edges(A, B, H, G, A)
% Inner square - counterclockwise
\Edges(C, E, F, D, C)
% Connect the corners
\Edge(A)(C)
\Edge(D)(B)
\Edge(H)(F)
\Edge(E)(G)

Figure 31 represents a flow network, with an augmenting path (of flow 4) highlighted. The source and sink nodes have been emphasized with different colors, and the augmenting path edges are wider than other edges and also of a different color. Observe that an edge may be curved using the bend right or bend left option when setting the edge style. Additional examples from the tkz-graph package can be seen in [9].

Andrew Mertz, William Slough and Nancy Van Cleave
% Change vertex attributes
\SetUpVertex[FillColor = gray!50, InnerSep = 5pt]

% Change edge attributes
\tikzset{LabelStyle/.style = {shape = circle, inner sep = 2pt}}
\Edge[color = gray!80, lw = 5pt, label = 16](s)(v1)
\tikzset{EdgeStyle/.append style = {bend right = 15}}
\Edge[lw = 2pt, label = 10](v1)(v2)

Figure 31: A sample flow network with changing attributes.

11 Summary
As educators in the field of computer science, we find ourselves challenged to produce a wide variety of figures and diagrams. Being able to replicate (or in some cases, exceed) the quality found in textbook presentations is a practical and intrinsically rewarding skill. Fortunately, the \TeX{} community has provided a wealth of resources which can be brought to bear on this problem. We hope that the examples and explanations provided in this paper will encourage others to explore these and other packages further.

References

⋄ Andrew Mertz, William Slough and Nancy Van Cleave
Department of Mathematics and Computer Science
Eastern Illinois University
Charleston, IL 61920
aemertz (at) eiu dot edu, waslough (at) eiu dot edu, nkvanclave (at) eiu dot edu

Typesetting figures for computer science