Image2Ascii conversion - State of the art

Overview

---

1. Preface

Working on my freeware ascii art tool JavE [0] for about three years now, I have gathered a lot of knowledge about ascii art and algorithms. JavE stands for *Java ascii versatile Editor* and the program now is a huge collection of algorithms concerning ascii art.

One of the trickiest modules in JavE is the image2ascii converter with its automatic conversion algorithms. Currently one can choose from about 9 different algorithms for conversion. As a lot of things can be adjusted, it should be no problem to spend some hours just playing around with the conversion options... I am asked quite often for information about those conversion algorithms and so I have decided to write this document. It started as a posting to a newsgroup (alt.ascii-art) in May 2002 and I try to extend it as I have time and inspiration.

If you have any questions or comments, feel free to contact me: markus@jave.de

2. Image2ascii conversion algorithms

2.1 Overview

As far as I know there are 3 kinds of i2a (image2ascii) algorithm categories:

- 1. Black/white-algorithms
- 2. Greyscale-algorithms
- 3. Edge-tracing or edge-detection algorithms

I am going to explain them and show you examples in the next sections.

2.2 Black/white-algorithms

Black and white algorithms are the most simple conversion algorithms. As far as I know they were the first ones being developed. Here is an example:

low resolution:	higher resolution:

##### #### ## ##### 	
  # # # #
  # ## # #
  # # # #
  # #### ## #
	88888 8888 d8b 88888
	  8 8]b 8
	  8 88 Yb 8
	  8 8 o d8 8
	  8 8888 YP 8

Low resolution b/w algorithms only use two characters representing black and white pixels. Higher resolution algorithms use more than one pixel for a character. I have not focused on b/w conversion, so the algorithms in JavE are not the best ones for it. Well, ok, I will explain a very simple b/w algorithm: "2by2 (JFL..)". It takes 4 pixels from the original image and converts them to black and white. Then it chooses the character from the following table containing all 16 patterns that can be built using 4 pixels:

Pattern:	.. / ..	.X / ..	X. / ..	.. / X.	.. / .X	XX / ..	.. / XX	X. / X.	.X / .X	.X / XX	XX / .X	XX / X.	XX / XX
Character:	`	'	.	,	"	_	(	)	J	L	7	P	8

The problem here are the characters '(' and ')', for they do not look that good in combination with the others. One could also use '[' and ']', but the quality of the result will much depend on the font actually used for presentation.

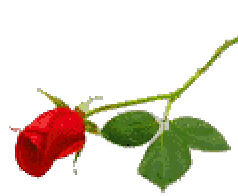
Note that there is also a tutorial about how they (and some simple greyscale algorithms) work in the www at [1].

2.3 Greyscale-algorithms

The main goals with grayscale algorithms are:

- The overall brightness of each character after conversion should be as close to the original image brightness of that region as possible. So as the font is made smaller and smaller or the distance between the eyes of the observer and the image increases the text will more and more look like a photo - although there is by far less contrast.
- Edges, outlines and other places with structure and high contrast on the image should be preserved using characters more or less fitting best to the structure in the image.

Greyscale algorithms try to find a character that has a brightness that is very close to the original pixel in the image. Most algorithms use 1 pixel to convert it to 1 character:



```
##@H@@@ku=;^^?:^;;;;:^...`````;p#Ny=HHH
NHHgH#fr;^;^.:lrrt?:^^^....``.;p#M@;m@H
NNHHHz?:.^:=uyZyybfyl?;^..^=Zbbk@NH=yH@
gHN@r:^..?kH@@Hb?pqzvl?^`?mHkuH##NHtZ@@
NMqr;^.;rvtZyZZ;^;tzvt?``rvulz@H##@zrkH
Mpt:.;Zut=????=?;;?ttr=.`:?lvyyg#H@Z?mH
yt^^fgZvt=?::^^::;?=ttt^ ^;?trzkHH@p;mN
t.;HHkurtl?;;::::;=ltrt; .;?ltu@#HHg?ZN
:vMNmqZrtl=??;;;;?trtvr?``??lrpNHgHH?tN
fNNNpkZvrtl=???;?=rruuut^^==tyHNH@HNr=@
#H##yfuvrtl===??;??=l??^^;=lZHNH@mH#p=b
```

```
                           `.^.`
                          .^
                        ..`
    `.    `       ``..^.`
    .?fr?^..^.^;;:` ``
``?yp@@@@b=.?yVfppZ^^lvrt?.
.@#@NHNNNV`.=pg@gfrtyVVyyyu=`
:NMMNHfrl.        ?yZyyy?.^^^.
 ?kHp.  `        .Zyfyfr.
                   ^=;.
```

"state of the art"-algorithms use more than 1 pixel per character and the result shows more details. Here are the same examples using a 4 pixel per character algorithm:

```
MMMMHNNM0C++^+;.++++:+.....`` ?WMMKjMMH
MMMHNMBZ+!+!^.+A&&++::..^^.....dMNN+HM@
N#NHMSC+``!JdAQmgkWHw++.^^.JXWmkH#MIdM@
mMMMZ+!`.JWMMMM#?WMwwOz` .HMHTMHHMMrdHH
MM#Z+!..wwOTHXU+.?Owzwz`.UZUCjHMMMHPdkM
NEZ!`.X0t=z111z1++1OtvI. ?1OwUWHMMMb?mH
EZ!.+MXvtzz;;::::+1lOrO!`.+1trwWH@Mb+HM
Z`.MMHXvrl=?;::;;+zlOvOz .+=lrwHHMM#+d#
:JNMHHXvrtl=1++;?1OwOzwz. ==trXMHgMNzvM
iMMMWHXzrtll=????zwwQAww+.=lOdMMMHMMZzM
M@MMXHXzvrtll=????=zOI1!`+zOQMHMMHMMbjU
```

```
                             .?!
                           .`
                        ..`
     .             ....``
    ..Q+.,...``+..
  .JWHHHHR+..dXXkWk,.Juu&..
 dNHMMMMM@ `"BHHHB6JXWWXXXX&.
.MMMMMB"7!        .XXXWX|`??!`
 7WMY^            XWWXWY'
                   ?7=!
```

Simple 1 pixel per character algorithms use a "greyscale gradient" or "greyscale ramp" defined by ordering all or only a few characters depending on their brightness. Here are two examples of greyscale gradients only containing a few characters:

.:OI#M
.:oO8@

I like the second one very much, because the characters are all more or less symetric and they do not contain straight lines.

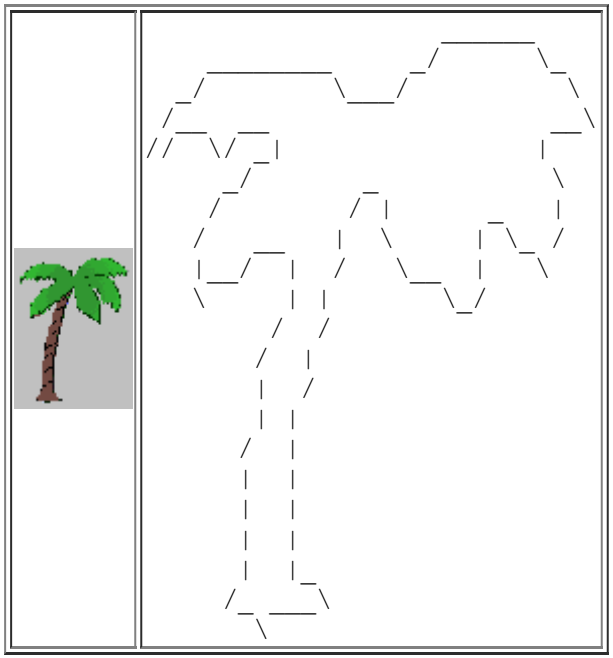
A more sophisticated strategy is to create a greyscale table containing brightness values for each character in a font. Most converters only have one greyscale table derivated from the font the author of the program has used when writing the algorithm. JavE however now contains about 20 character tables containing greyscale values for the 1 and 4 pixel algorithms for different fixed width fonts.

For converting to a bigger result (>100 characters wide) it is a good idea to also use features like error diffusion (with 1-pixel algorithms) or to combine both of the above algorithms (and maybe to use some other tricks). An algorithm doing that can be found in version 5.0 (or greater) of JavE. As with that I believe that the limit of what can be achieved with pure greyscale algorithms has be reached.

2.4 Edge-tracing or edge-detection algorithms

Many ascii art fans believe edge detecting algorithms will some day lead automated image conversion to a breakthrough. Well, I do not (yet) think so... Those algorithms try to detect edges and lines in images by using more or less complicated algorithms and then convert the lines to ascii.

In fact I only know about two of those algorithms. There is a very simple one for edge tracing, but it is so simple that it is more like cheating than a *real* algorithm:

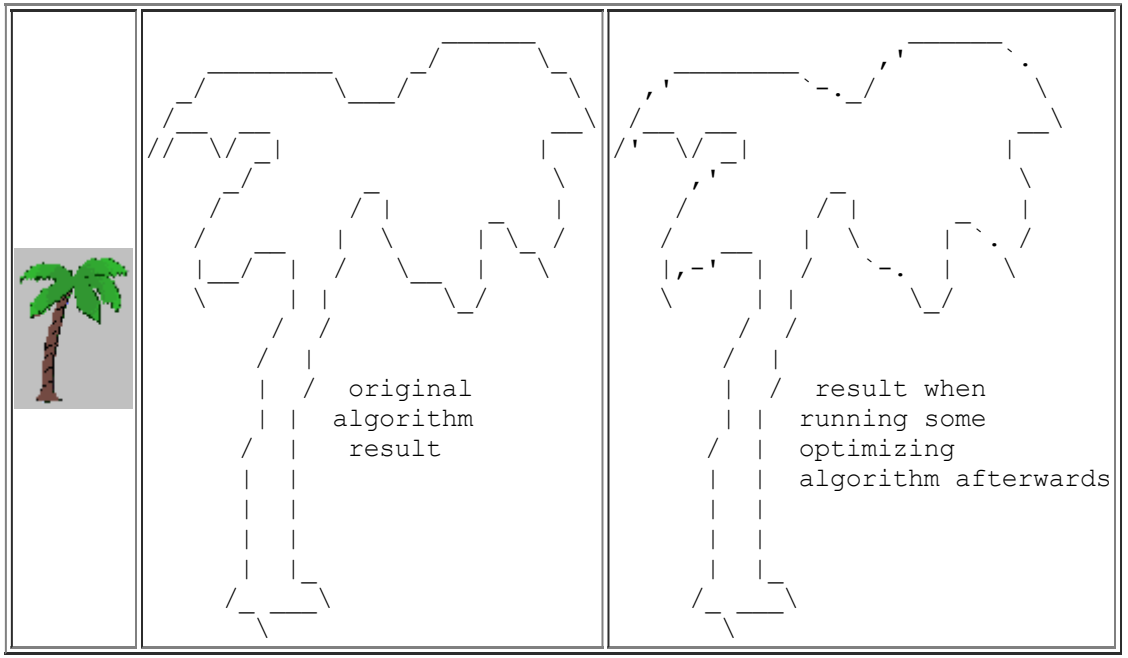


The basic algorithm once could be found here: [3]. However the site containing the information seems to be removed.

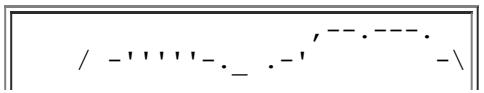
It is not hard to find out how the algorithm works. Here is the pseudocode, I have once posted it to the newsgroup along with more demo images [4]:

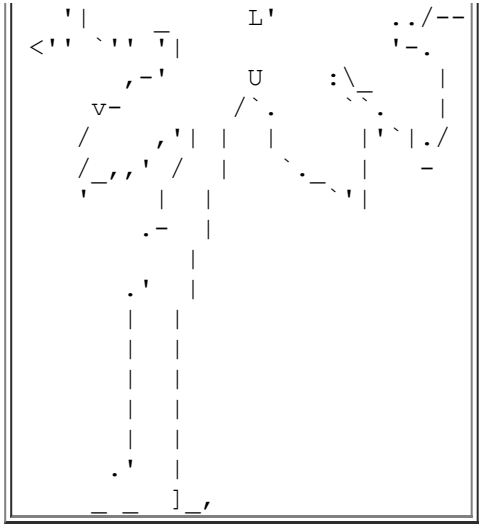
```
FOR ALL PIXELS x,y IN THE IMAGE DO:
  IF IS_SET(x,y)
    IF NOT_SET(x-1,y) AND NOT_SET(x,y-1)      PRINT /
    ELSEIF NOT_SET(x+1,y) AND NOT_SET(x,y-1) PRINT \
    ELSEIF NOT_SET(x-1,y) AND NOT_SET(x,y+1) PRINT \
    ELSEIF NOT_SET(x+1,y) AND NOT_SET(x,y+1) PRINT /
    ELSEIF NOT_SET(x,y+1)                    PRINT _
    ELSEIF NOT_SET(x-1,y) OR NOT_SET(x+1,y)  PRINT |
    END
  ELSEIF IS_SET(x,y+1) AND IS_SET(x-1,y+1) AND IS_SET(x+1,y+1) PRINT _
  END
END
```

As JavE also contains some optimizing algorithms for line drawing I have managed to slightly improve the result by some kind of filtering the ascii art after conversion:



Finally there is a very complicated *real* edge detection algorithm:





Line detection works as described in many books about image processing. For converting lines to ascii art the algorithm then uses JavE's capabilities for drawing lines. The conversion resolut is not really as good as expected, but with some images it is a good point to start from for editing by hand. (Read more about edge detection algorithms and combination with grayscale algorithms in the google groups archive at [2])

2.5 Conclusion

Any algorithm I have seen can more or less be assigned to one of these three types. However there are also some ideas for different strategies. If you have investigated more about one of them let me know.

2.6 Touching up the conversion result

As you can see by the examples above, it is always a good idea to do a bit of work by hand to improve the conversion result. There is a nice tutorial that explains how to use the autmatical conversion to start from to creating *real* art at [5].

I think (for beginners) it is a good idea to convert an image trying to find a good algorithm. Then paste the result into your favorite text editor and make changes by hand until it looks best.

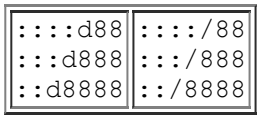
2.6 Plastic bag technique

A great way for converting an image by hand is using the plastic bag technique: The original image is placed in the background of your text editor (as kind of watermark) and you can draw the Ascii image by hand. The required watermark functionallity is not

included in many editors. JavE [0] however does support it.

2.7 Further investigations

There have been some investigations on combining edge detection and greyscale algorithms. But this does not seem to work well. Look at this simple example:



The diagonal lines from the edge detection also generate empty triangles. This effect becomes even worse when dealing with details like eyes for example. There is an idea for solving this problem: One could define rules for mixing the characters from both conversions. There is this kind of mixing algorithm in JavE:

```
- + | = +
o + d = d
/ + \ = X
```

But this algorithm is only defined for about 200 combinations and there are about 90x90=8100. Also first tests have shown that it does not work that well [4]...

(BTW: More often than once I have considered creating a very simple plugin architecture for JavE, so that everybody with basic Java programming skills can easily write his/her own conversion algorithm, w/o having to care about loading, scaling and preprocessing images etc. Is there any need for this or is implementing it just wasted time?)

A. Links:

[0] http://www.jave.de
[1] http://vyznev.net/ascii/ada/aaatechnical.txt (have a look at the last third of that text)
[2] http://groups.google.com/groups?selm=3C59AF2E.92AC8B89%40rz.uni-karlsruhe.de&oe=utf-8&output=gplain
[3] http://www.ai.mit.edu/~jsd/Projects/ImagesToText/
[4] http://groups.google.com/groups?selm=3C5318B5.80F13461%40rz.uni-karlsruhe.de&oe=utf-8&output=gplain
[5] http://www.jave.de/docs/tutorial2/index.html

---

last modified: 8th January 2005