Structure-based ASCII Art

Eduardo Magnus Lazuta[1], **Rayan Raddatz de Matos**[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

{emlazuta,rayan.raddatz}@inf.ufrgs.br

Abstract.

1. Introdução

A incapacidade de computadores antigos de realizar representações gráficas sofisticadas deu origem a um novo tipo de arte digital: Arte em ASCII. Essa forma de expressão artística, chamada de ASCII Art, utiliza o padrão de caracteres ASCII [Cerf 1969] para recriar imagens utilizando símbolos textuais. Apesar do nome popular ASCII Art, outros padrões de texto, como UTF-8, são utilizados atualmente para permitir artes mais detalhadas e expressivas. Além disso, mesmo com a modernização dos sistemas computacionais e o avanço das representações gráficas nesses sistemas, a ASCII Art continua popular, principalmente em comentários de plataformas de vídeo e streaming.

Esta forma de arte busca criar ou replicar imagens por meio de duas principais abordagens: com base no tom ou na estrutura. Enquanto a ASCII Art com base no tom busca mimetizar os tons presentes na imagem com diferentes caracteres representando diferentes intensidades, a ASCII Art com base na estrutura busca compreender e adaptar a estrutura da imagem para poder transportar essa estrutura para os caracteres computacionais. Ao pensar de forma extrema, exemplos de ASCII Art baseada em estrutura são: `:)` (sorriso), `T_T` (choro) e `S2` ou `<3` (corações). Representações essas utilizadas normalmente em aplicativos de mensagens, as quais buscam expressar a estrutura de formas ou feições visuais através de caracteres textuais. Assim, a utilização de ASCII Art baseada em estrutura permite criar representações fidedignas com uma menor quantidade de caracteres do que a baseada em tom.

Nesse trabalho, buscamos reproduzir os resultados obtidos em [Xu et al. 2010]. Esse trabalho introduz uma nova métrica chamada ”Alignment-Insensitive Shape Similarity”que, diferente das métricas da época, tolerava o desalinhamento das formas enquanto levava em conta diferenças na posição, orientação e escala. Além disso, eles propõem a criação da ASCII Art baseada em estrutura como uma otimização que visa minimizar a dissimilaridade de formas e deformação

Disponibilidade de Software e Dados. Nossos resultados são públicos. Disponibilizamos o material aqui mostrado através de um repositório público do GitHub em https://github.com/rddtz/ascii-art. Nosso material complementar contém o código fonte deste relatório, bem como o software e as imagens necessárias para recriar os exemplos. Também incluímos instruções para executar a aplicação.

2. Fundamentação Teórica e Métodos Utilizados

2.1. Histograma Log-Polar

Trata-se de um descritor robusto escolhido especificamente pela sua tolerância a desalinhamentos. Diferente dos histogramas convencionais cartesianos, este método emprega um sistema de coordenadas polares onde os bins (compartimentos de acumulação) são particionados uniformemente no espaço logarítmico.

O histograma log polar é um descritor altamente sensível a posições de pixels próximos ao centro (foco), mas torna-se progressivamente tolerante a variações espaciais conforme a distância aumenta. Além disso, ao acumular a densidade de pixels dentro de cada bin, o método torna-se inerentemente insensível a pequenas perturbações de forma, permitindo que caracteres ASCII representem fielmente a estrutura da imagem mesmo com pequenas transformações de escala ou posição.

Para gerar um descritor robusto com o histograma log-polar é preciso amostrar N pontos da imagem. Ao final do processo, os N histogramas log-polares são concatenados, formando o descritor final utilizado para representar a imagem.

2.2. Simulated Annealing

Para obter uma melhor classificação da imagem, é necessário permitir pequenas deformações controladas na representação atual. Para isso, utiliza-se a meta-heurística *Simulated Annealing*, que admite novas soluções de forma probabilística. Quando a solução gerada não é melhor que a atual, ela ainda pode ser aceita com probabilidade:

$$p = e^{-\Delta/T},$$

onde T é a temperatura da iteração e Δ é a diferença entre a energia da solução atual e a da nova solução.

A meta-heurística opera da seguinte forma: para cada temperatura, são realizadas N iterações, nas quais uma nova solução é gerada por meio das deformações local e global da imagem, buscando reduzir o valor da energia total. Ao final desse ciclo de N iterações, a temperatura é atualizada conforme a equação:

$$T = 0.2\, t_a\, c^{0.997},$$

onde t_a é o erro médio inicial e c corresponde ao índice da iteração.

O critério de parada do processo ocorre quando não há redução na energia da solução ao longo de N iterações consecutivas em uma mesma temperatura, indicando que o algoritmo convergiu para uma configuração estável.

Gerando a nova solução

2.3. Função Objetivo

A função objetivo mede a qualidade final do resultado combinando dois fatores: a *dissimilaridade visual* entre cada célula da imagem e o caractere escolhido (D_j^{AISS}), e a *deformação aplicada* à célula (D_j^{deform}). Ela é definida como:

$$E = \frac{1}{K} \sum_{j=1}^{m} D_j^{AISS} \cdot D_j^{deform}$$

onde m é o número total de células e K é o número de células não vazias. Quando não há deformação ($D_j^{deform} = 1$), a energia depende apenas da semelhança visual. Como a fórmula é normalizada por K, valores de E podem ser comparados entre diferentes resoluções de texto: quanto menor o valor de E, melhor e mais agradável é o resultado visual.

3. Implementação

Implementamos nossa solução utilizando a linguagem de programação `python` por sua simplicidade e boa integração com a biblioteca `OpenCV`. Essa seção destaca e descreve alguns pontos específicos da implementação.

3.1. Workflow

A Figura 1 sintetiza o fluxo de trabalho padrão da aplicação. Recebemos como entrada uma imagem em um formato qualquer. A partir disso ”esquelitizamos”e vetorizamos a imagem. O processo então segue para a otimização, onde buscamos minimizar a função objetivo. Quando alcançamos um resultado satisfatório com a otimização, passamos então para a classificação, onde designamos os caracteres que irão compor a imagem de saída.

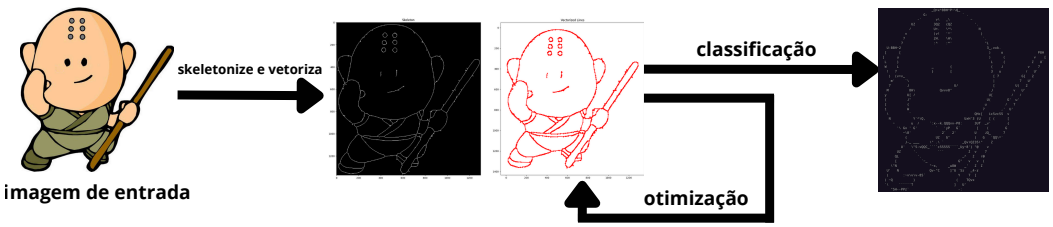


Figura 1. Workflow do método

3.2. Vetorização da Imagem

O artigo referência não deixa claro como é feita a vetorização das imagens, e faz alusão ao fato de que a entrada é composta pelos vetores que compõem a imagem alvo. Essa

simplificação não foi integrada ao nosso trabalho. Ao invés disso, adicionamos uma forma de vetorização de imagens genéricas através de dois diferentes modos: Utilizando ou a biblioteca OpenCV2 ou uma implementação manual baseada no algoritmo DFS.

Primeiro começamos esqueletizando a imagem através da função `skeletonize` da biblioteca skimage. Essa função faz com que todas as linhas da imagem tenham apenas 1 pixel de largura. Partindo dessa imagem esqueletizada, o método de vetorização que utiliza OpenCV2 faz uso da função findContours. Essa função contorna todas as linhas da imagem, e, como isso gera linhas duplicadas, utilizamos apenas metade dos pontos gerados. O método que utiliza o DFS funciona criando linhas ao ligar os pontos vizinhos. Ao termos as coordenadas de todos os pixels que formam uma determinada linha nós utilizamos a função `approximate_polygon` da biblioteca skimage para gera uma linha através desses pontos.

3.3. Otimização e Geração da Imagem

Algumas das fórmulas utilizadas na otimização e classificação não são explicitamente detalhadas no artigo referências, ou muitas vezes são expostas de forma ambígua. A parte da classificação se mostrou simples quando comparado com a otimização. Tentamos seguir as fórmulas descritas no artigo a risca, entretanto para a parte da otimização algumas etapas não foram completamente esclarecidas no artigo referência, principalmente ao se tratar da restrição de acessibilidade e do cálculo da energia. Isso dificultou a implementação, e foi necessário ir atrás de códigos referências e outras implementações e discussões sobre o assunto em busca de entender os métodos mencionados. Uma estratégia que se mostrou interessante para melhorar a acuidade visual dos resultados foi diminuir a quantidade de caracteres disponíveis, utilizando apenas algumas letras e símbolos selecionados para a criação da imagem. Por fim, a otimização se mostrou uma etapa custosa, principalmente para nossa implementação em python, por isso não foram seguidas a riscas condições de paradas estabelecidas no artigo referência.

4. Resultados

A Figura 2 mostra alguns resultados obtidos, o tamanho final da imagem é um grande fator da qualidade. O processo de otimização demonstrou pouco resultado em razão do grande tempo de computação. Entretanto mesmo sem aplicar a otimização o resultado final já é final agradável, como podemos ver nos exemplos.

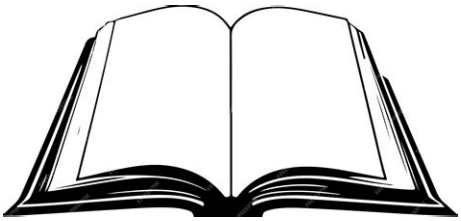
5. Conclusão e Trabalhos Futuros

Conseguimos observar que os resultados gerados pela aplicação são satisfatórios mesmo antes da aplicação da otimização. Um fato que possa ter impedido a eficácia da otimização seja a alta quantidade de pontos gerados pelo processo de vetorização automática. Melhorias futuras podem focar em melhorar o processo de vetorização e otimização, além de poder utilizar métodos mais atuais com redes neurais para classificação dos caracteres.

Referências

Cerf, V. G. (1969). Ascii format for network interchange. *RFC 20*. `https://www.rfc-editor.org/rfc/rfc20.html`.

Xu, X., Zhang, L., and Wong, T.-T. (2010). Structure-based ascii art. *ACM Trans. Graph.*, 29(4).

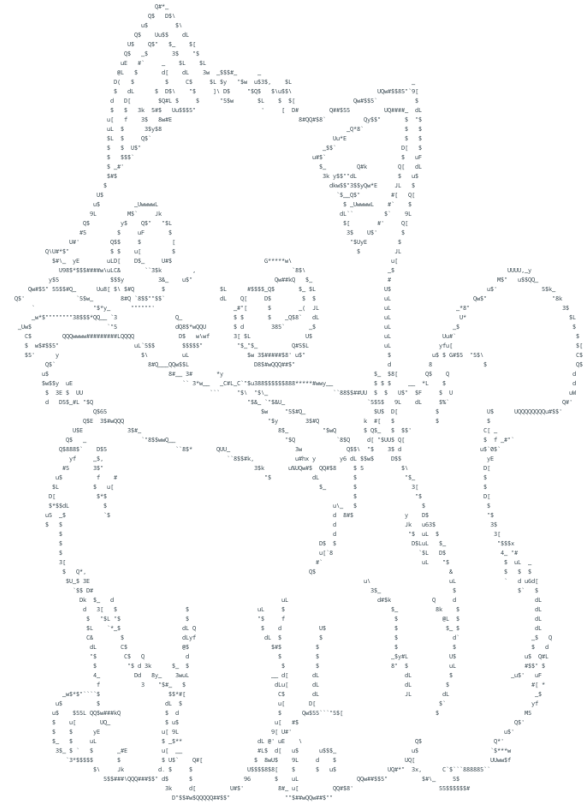


(a) Livro Original

(b) Livro ASCII com 100 colunas

(c) Gato Orignal

(d) Gato ASCII com 100 colunas



(e) Gato ASCII com 200 colunas

Figura 2. Resultados da transformação de imagens para ASCII sem otimização.