Intelligent Scissors Write Up

David Ye

July 18, 2023

1 Edge Weight Model

I tested intelligent scissors on sample.png (the blue bird).

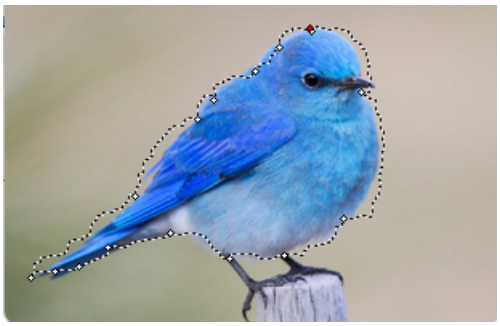

Figure 1: Cutting out blue bird using original intelligent scissors with 16 anchors

In figure 1, I cut the blue bird out using 16 anchors. Unfortunately, the cut turned out very rough and did not closely follow the bird's outline. There are small buffers between most of the cuts and the bird's body. Interestingly, the cuts underneath the bird's tail followed the bird very nicely. In figure 2, I decreased the number of anchors to five, and the cut turned out much worse. There are large buffers between the cut and the bird's back and tail. The scissors also cut through the bird's belly rather than around it.

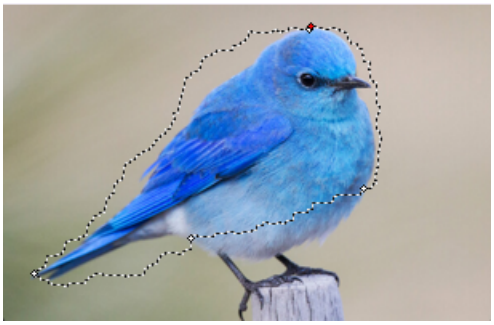

Figure 2: Cutting out blue bird using original intelligent scissors with five anchors

Interestingly, when the Z-crossing score is calculated opposite of the specification (set Z-crossing score to be one when the absolute value of the Laplacian is smaller than its neighbor and zero

otherwise), the intelligent scissors improved greatly. In figure 3, the scissors nicely cut around the bird using just four anchors, which is a huge improvement from before.

Figure 3: Cutting out blue bird using modified intelligent scissors with four anchors

To find out why the modified scissors worked better than the original, I plotted cost matrices of the bird. Figure 4 shows the cost matrix using the original Z-crossing score, and figure 5 shows the cost matrix using the modified Z-crossing score. In both figures, pixels with costs closer to zero appear white and pixels with costs closer to one appear black.

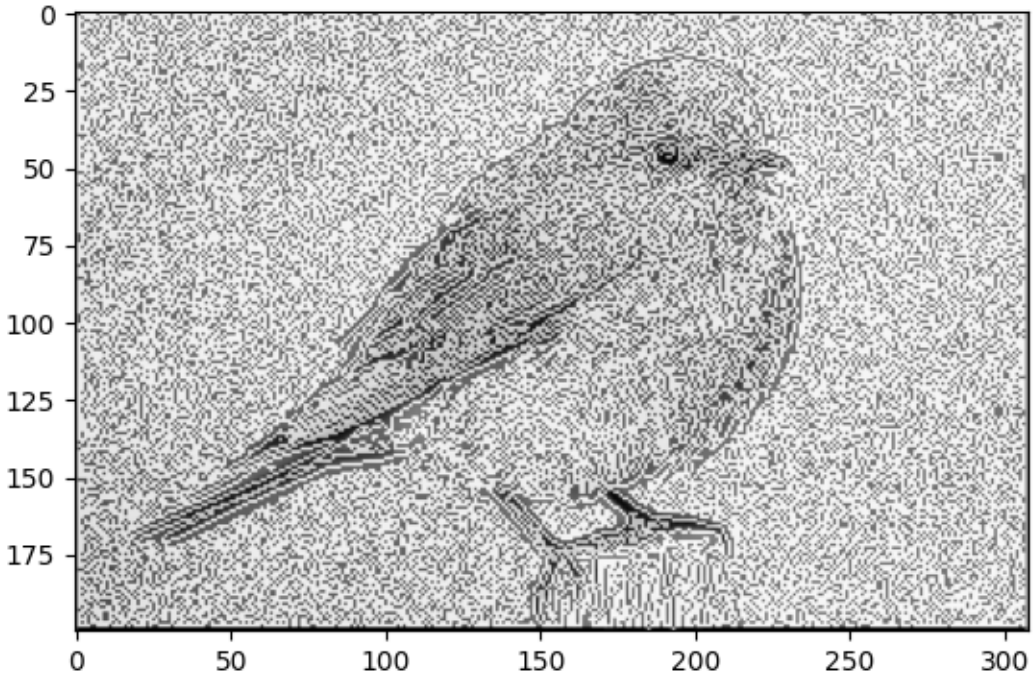

Figure 4: Cost matrix of blue bird using original Z-crossing score

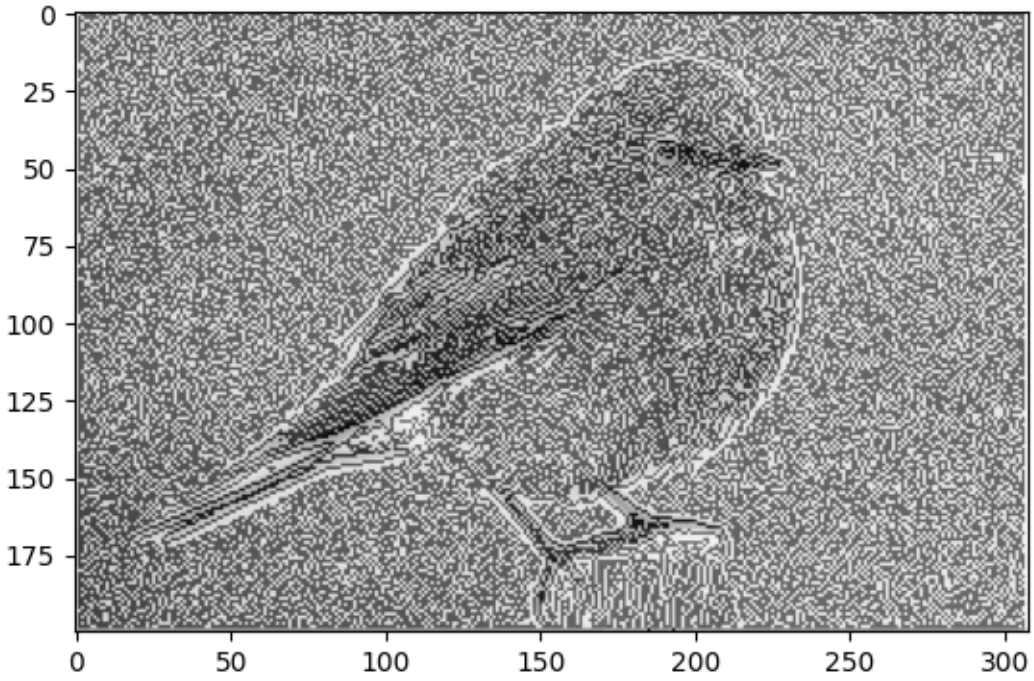

Figure 5: Cost matrix of blue bird using modified Z-crossing score

In figure 4, there is thin one pixel white edge (zero cost) around the bird. In theory, this would be an excellent cost matrix since the edge surrounding the bird is so sharp and precise. However, since the edge is only one pixel in width, the edge cannot be adjacent/orthogonal neighbors with itself along diagonals. To follow a diagonal, the edge may be diagonally connected with itself as seen in 6 (diagonal edge pixels marked red). Since diagonal pixels are not directly connected, the only way to get from one edge pixel to another diagonal to it is by traveling horizontally/vertically to a new pixel and then vertically/horizontally to the edge pixel. During this process, there may be other non-edge pixels that are close by that have a lower cost to travel to. Thus, the one pixel white edge will be derailed and led astray as seen in figure 1.

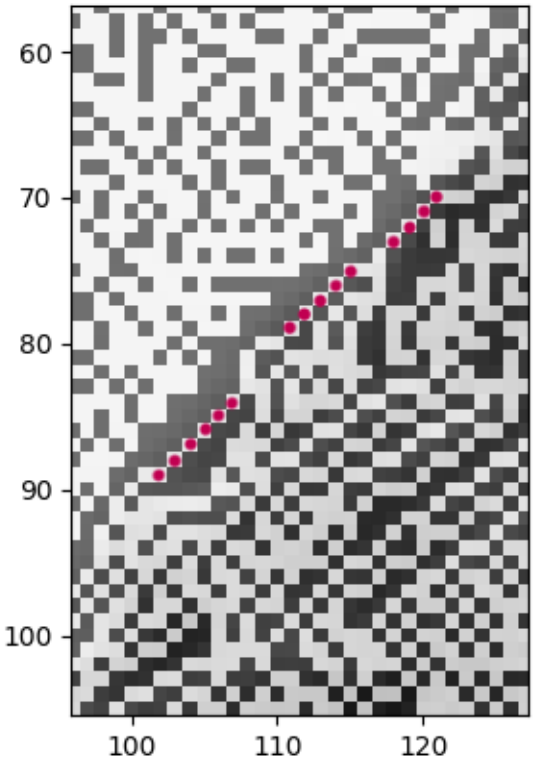

Figure 6: Zoomed in on cost matrix of blue bird's back using original Z-crossing score, parts of the one pixel white edge that are diagonal to each other are marked in red

The theory about is shown in practice in figure 7 and figure 8 where in both cases the scissors work much better cutting horizontally compared to cutting diagonally.

Figure 7: Original intelligent scissors on blue bird's back, one the left the back of the bird is rotated to horizontal position, while on the right bird is left in the original orientation.

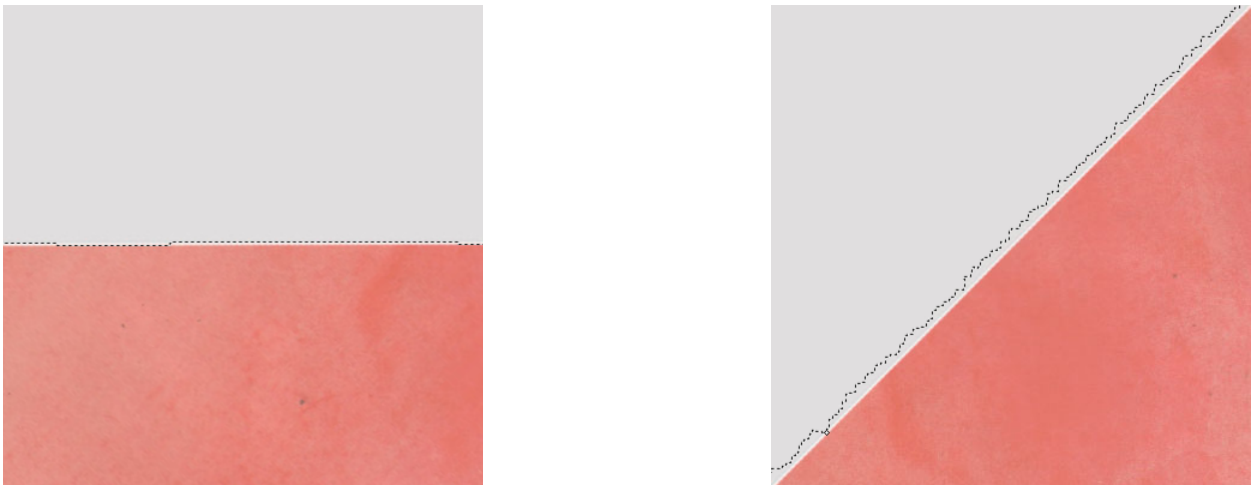

Figure 8: Original intelligent scissors on grey/red plane. On the left the planes intersect horizontally, and on the right the planes intersect diagonally.

On the other hand, in figure 5 (cost computed using modified Z-crossing score), there is a thin one pixel black edge (cost of one) around the bird. Closely surrounding the thin black edge is a much thicker white edge. The white edge closely follows the birds outline and is much more continuous than in figure 4. The increase in thickness of the white edge makes it much more robust to the noise inside and outside the bird. This robustness boosts the performance of the scissors by supporting continuous cuts that follow the outline of the bird as seen in figure 3.

At first glance, it seems like the inverted implementation of the Z-crossing score does it all. However, taking a closer look, it does not precisely follow the outline of the image. Instead, it "outlines the outline" of the image. However, more problems come into play when there is a plain background. Since the Z-crossing score is inverted, the plain background will have an extremely low cost (due to zero Z-crossing score) as seen in figure 11. This will misguide the scissor to cut around the background, rather than around the image as seen in figure 12.

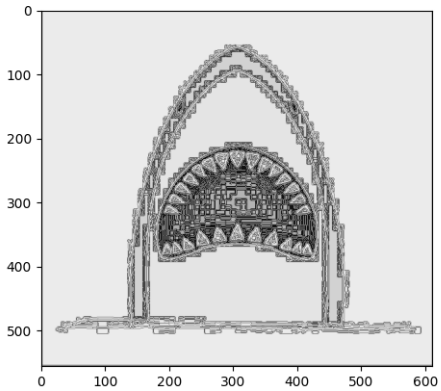

Figure 9: Cost matrix of shark using modified Z-crossing score

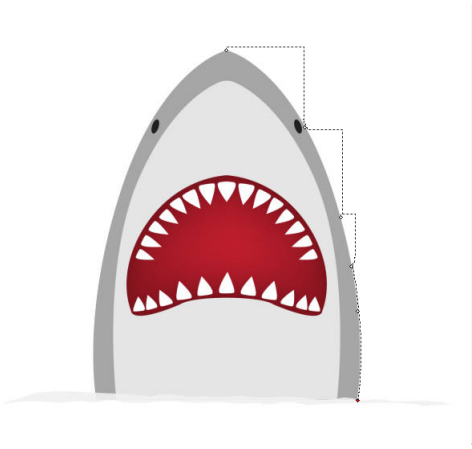

Figure 10: Cutting out shark using modified intelligent scissors

Now if we use the original implementation of the Z-crossing score, the cost of the plain background is much higher (due to Z-crossing score of one) as seen in figure 11. Thus, the original implementation is much better at tracing the shark as seen in figure 12.

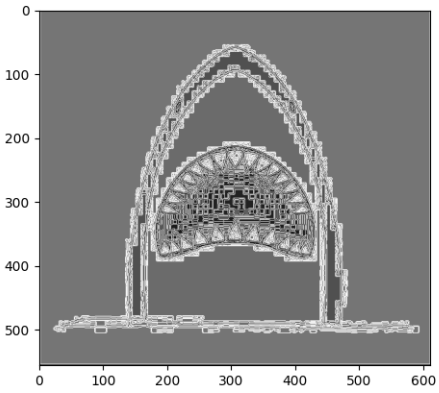

Figure 11: Cost matrix of shark using original Z-crossing score

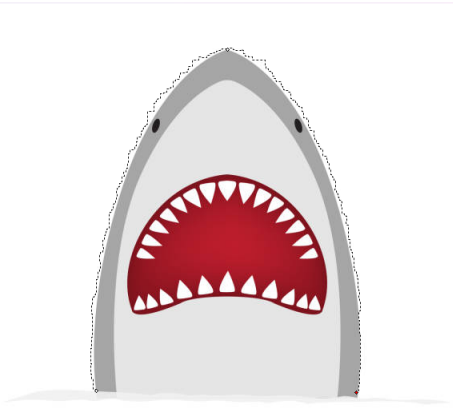

Figure 12: Cutting out shark using original intelligent scissors

We have seen that both implementations of intelligent scissors has its flaws. The original/correct implementation of intelligent scissors is extremely bad at handling curves in images due to the single pixel edge's inability of connecting directly with diagonal edge pixels. However, the reason the user wants to use intelligent scissors is because the object being cut out is not a horizontal/vertical cut, but rather has many curves that makes hand cutting it tedious. As a result, the current implementation of edge weights is not good enough. It could be a good idea to consider diagonal pixels as neighbors that are directly connected. This would increase the scissor's ability to handle curves.

In the meantime, the modified intelligent scissor with the inverse Z-cross scoring appears to handle curves much better. However in some cases where the background is very plain, the scissors will fail spectacularly. Despite these risks, this modified intelligent scissor works fairly well for most images I tested.

A weakness that appears in both implementation is their ability to handle shadows and highlights in images. The intelligent scissors tend to follow the highlights/shadows rather than the actual object.

2 Naive vs. Optimized Dijkstra's

To test the speedup of optimized vs naive Dijkstra, I randomly selected a starting vertex and ran Dijkstra till all shortest paths were found. The speedup in practice is does not match up with the improvement of $O(|V|^2)$ versus $O((|V| + |E|) \log |V|)$ as seen in table 1.

Image	Optimized (s)	Naive (s)	In Practice (xFaster)	In Theory (xFaster)	Total Vertices
sample	0.292	6.172	21.108	1294.098	61600
french bulldog	1.114	45.898	41.175	4482.332	240000
bear cub	0.208	4.383	21.006	1077.281	50325
sleeping fox	1.219	63.107	51.727	4756.210	256000
deer	21.201	1729.717	81.583	41706.655	2670000

Table 1: Comparing in-practice speed up to theoretical speedup assuming a theoretical improvement of $O(|V|^2)$ versus $O((|V| + |E|) \log |V|)$.

The reason is because the $O(|V|^2)$ time complexity is the worst case scenario, where each vertex is connected to all the other vertices. Only in this case, will it take approximately $O(|V|)$ operations to find the minimum cost vertex. In our situation, each vertex is only connected to neighboring vertices adjacent/orthogonal to it. Thus, the number of vertices in **todo** is approximately equal to the number of vertices that border the pixels that have been locked-in. In figure 15, all the vertices that have been visited are darkened, and the pixels that lie on the very edge of the darkened area represent pixels currently in **todo**. Figure 14 shows all the vertices currently in **todo** in red.

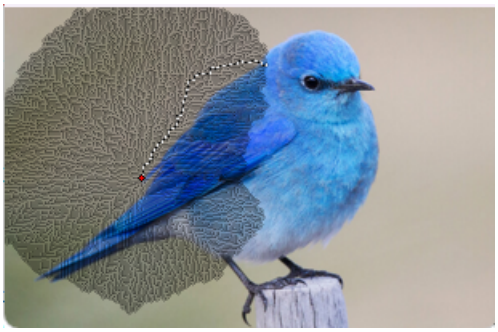

Figure 13: Darkened pixels represent pixels that have been visited by Dijkstra's. The pixels that lie on the very edge of the darkened area are the pixels currently in **todo**

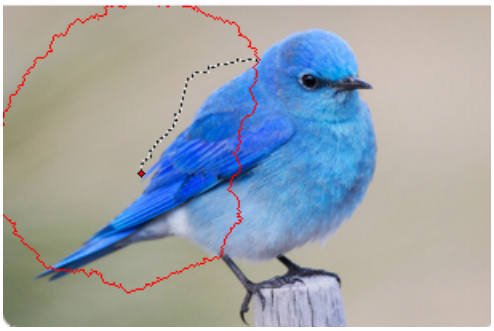

Figure 14: The red border are the pixels currently in **todo**

Since the largest border possible in our scenario is essentially the image parameter, a much better estimate of worse-case scenario time complexity is $O(V * (2 * W + 2 * H))$ where W = width of image in pixels and H = height of image in pixels. The improvement of $O(V * (2 * W + 2 * H))$ versus $O((|V| + |E|) \log |V|)$ matches up fairly well to the speedup obtained in practice as seen in the table 2. The reason they do not match up exactly is likely due to CPU spikes from other applications

running on the computer. Furthermore, the time complexities calculated are approximate, and may be off by some coefficient or constant.

Image	Optimized (s)	Naive (s)	In Practice (xFaster)	In Theory (xFaster)	Total Vertices
sample	0.252	5.163	20.472	21.344	61600
french bulldog	1.154	44.387	38.453	37.352	240000
bear cub	0.235	5.126	21.762	19.608	50325
sleeping fox	1.298	57.961	44.631	38.644	256000
cute animal	3.603	179.078	49.696	56.199	647513
cute puppy	1.343	57.533	42.808	38.694	261042

Table 2: Comparing in-practice speed up to theoretical speedup assuming a theoretical improvement of $O(V * (2 * W + 2 * H))$ versus $O((|V| + |E|) \log |V|)$.

3 Miscellaneous

Another optimization took place in the image_to_graph function definition. I realized for large images, calculating the graph representation of the image took a very long time. I optimized this by calculating the horizontal and vertical edge weights outside the for-loop using matrix operations. And inside the forloop, each pixel only looks at the pixel to the right and the pixel beneath it, rather than all four neighbors. This minimizes the repetitiveness of examining the same edge weights multiple times.

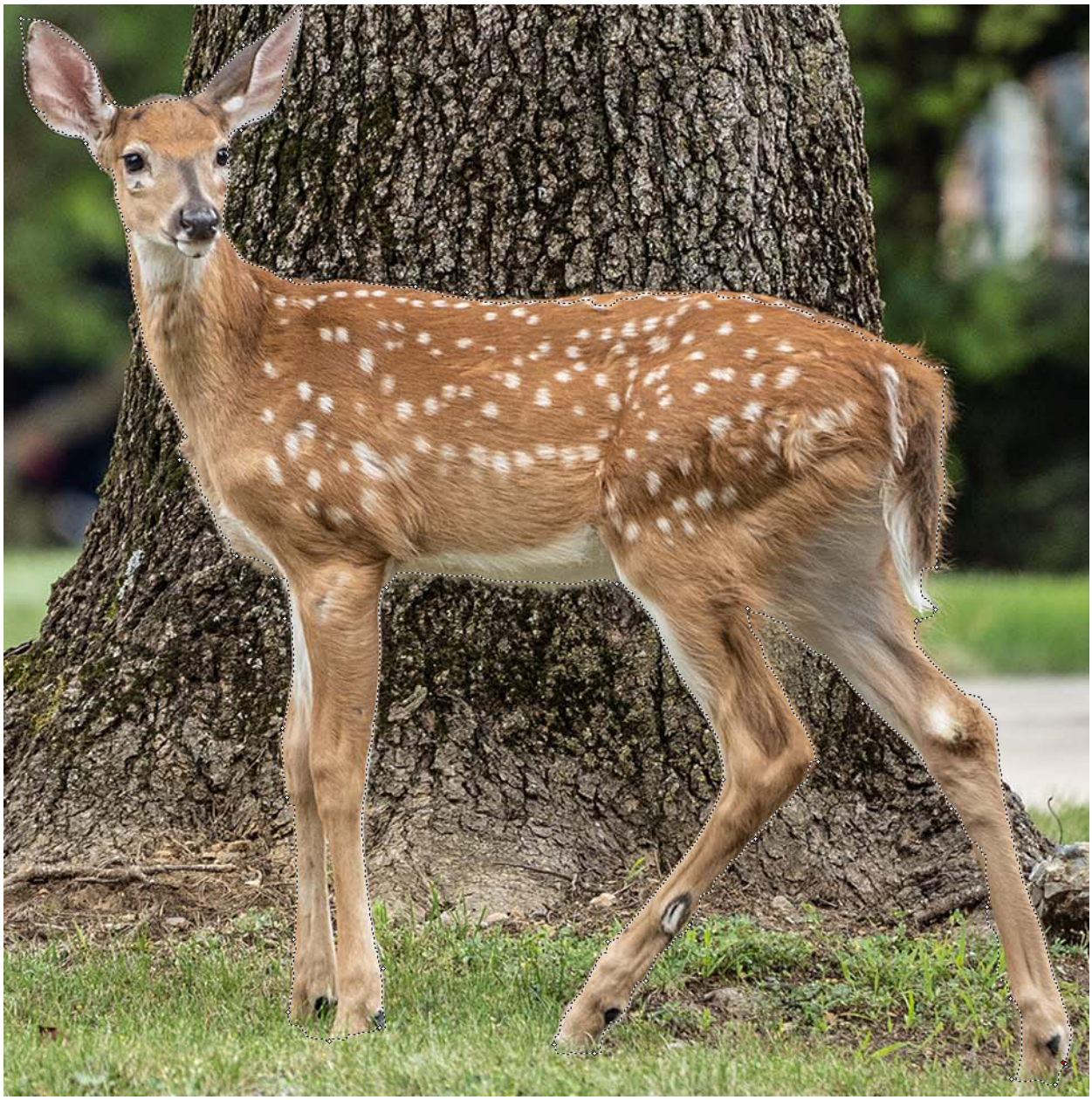

Figure 15: Thanks for reading!