In the BFS finder, at each iteration, we must project the gradient into the nullspace of the active constraints. Currently, this is accomplished by constructing a matrix $\mathbf{p}$ that describes the active constraints (described in detail below) and then using the SVD of $\mathbf{p}$ to find a basis for the nullspace of $\mathbf{p}$. This is highly inefficient because $\mathbf{p}$ is sparse and nearly orthgonal. We also do not need to compute a full basis of the nullspace but rather project the gradient away from the rows of $\mathbf{p}$. $\mathbf{p}$ has n^2 columns. We describe an efficient method to perform this projection.

1. The main loop of the algorithm requires at most n^2 steps. At each step, at least one more constraint becomes active (that is one value of $\hat{\mathbf{X}} = \mathbf{UY}$ becomes ± 1-valued). Suppose at step i, there are t active constraints, and at this step we only add one new constraint. Specifically, suppose $\hat{x}_{jk}$ becomes active. In this case, we add the following row to $\mathbf{p}$:

$$\mathbf{p}_t = \begin{bmatrix} \mathbf{0}_{i*n} & y^{(j)\intercal} & \mathbf{0}_{(n-i-1)*n} \end{bmatrix}.$$

   That is, the tth row of $\mathbf{p}$ will consist of $i * n$ zeros, followed by the jth column of $\mathbf{y}$, and the remainder of the row is zeros. If multiple constraints become active in one step, just add multiple rows. If we get multiple new constraints from different values of i, then we can do most of the following loop in parallel for each new constraint.

   (a) We may want to instead keep $\mathbf{p}$ as a block-diagonal matrix. So it'd look something like

$$\mathbf{p} = \begin{pmatrix} \mathbf{F} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{G} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{H} \end{pmatrix}.$$

   At each step, we must simply insert $\mathbf{p}_t$ into the appropriate row of $\mathbf{p}$ and keep a separate data structure that tells us how many rows each block contains so we don't have to sort through $\mathbf{p}$ everytime we want to do an insert. We should also be able to have $\mathbf{p}$ be preallocated and just store a dictionary of rows so we don't have to copy and move rows of $\mathbf{p}$ for each update.

2. Having updated $\mathbf{p}$, we next need to update the product $\mathbf{A} = \mathbf{pp}^\intercal$. Because of the block structure structure of $\mathbf{p}$, we should only have to multiply one block of $\mathbf{A}$ by the new row $\mathbf{p}_t$ and so this step should only require at most $n * k$ multiplications. Also notice that $\mathbf{A}$ is symmetric. (Possible optimization: we don't really care about the diagonal values of $\mathbf{A}$ either at this point unless you want to use them later in Step 4).

3. We next need to find each off-diagonal entry of $\mathbf{A}$ that is non-zero. Really you should just save off the values in the previous step when you perform the multiplication. The non-zero off-diagonal entires are called 'bad'. At most, we can have k bad entries per new row, but typically we have zero or two bad entries at least for $n = 4$.

4. We now use a modified version of the Gram-Schmidt proceedure to find an orthonormal basis for the space spanned by $\mathbf{p}$. For each block:

   (a) For each pair of bad entires, let (i, j) denote a pair and do:

      i. $u = p_i$, $v = p_j$
      ii. $v = v - \frac{uv^\intercal}{uu^\intercal}u$. (Possible optimization: $\mathbf{uu}^\intercal$ has already been computed or is equal to one if i has appeard before in this loop. Keeping track of this might actually be more work than recomputing it here).
      iii. Compute $\|v\| = vv^\intercal$. If $\|v\| > 0$, set $p_j = \frac{v}{\|v\|}$, otherwise p_j is a redundant constraint, save this index for removal and leave p_j as is for now.

   (b) Remove all redundant rows of $\mathbf{p}$ found in the previous loop

5. Set $\mathbf{A} = \mathbf{I}$ (this is guaranteed by the previous step).

6. Now we must project $\mathbf{V}$ into the nullspace of $\mathbf{p}$. Since $\mathbf{p}$ is now an orthogonal basis, this can be accomplished by performing a vector rejection on each row of $\mathbf{p}$. Explictly, this is done as follows:

   (a) $\mathbf{v} = \text{vec}(\mathbf{V})$.

   (b) For each row $\mathbf{p}_i$ in $\mathbf{p}$:

      i. $\mathbf{v} = \mathbf{v} - (\mathbf{v} \cdot \mathbf{p}_i^\intercal)\,\mathbf{p}_i$. Note that in a normal vector rejection, we divide by the norm of $\mathbf{p}_i$, but we have already normalized so $\|\mathbf{p}_i\| = 1$.

7. We now continue as usual, moving in the direction of $\mathbf{V}$ until a new constraint becomes active.

As a crude upperbound, we require n^2 iterations of the outerloop, and for each constraint, we can find at most k bad rows of $\mathbf{p}$. In this case, the orthonormalization will take $O(nk^2)$ operations per iteration making the overall runtime something like $O(n^3k^2)$. However, we will often find only two bad entries per iteration, in which case orthonormalization only requires $O(n)$ and so we can lowerbound the runtime as $O(n^3)$. I'm optimistic we get to the point that computing the inverse of $\mathbf{U}$ at each step becomes the most expensive part of the BFS solver. If we get to that point, then I have some ideas on how to simplify that operation as well.