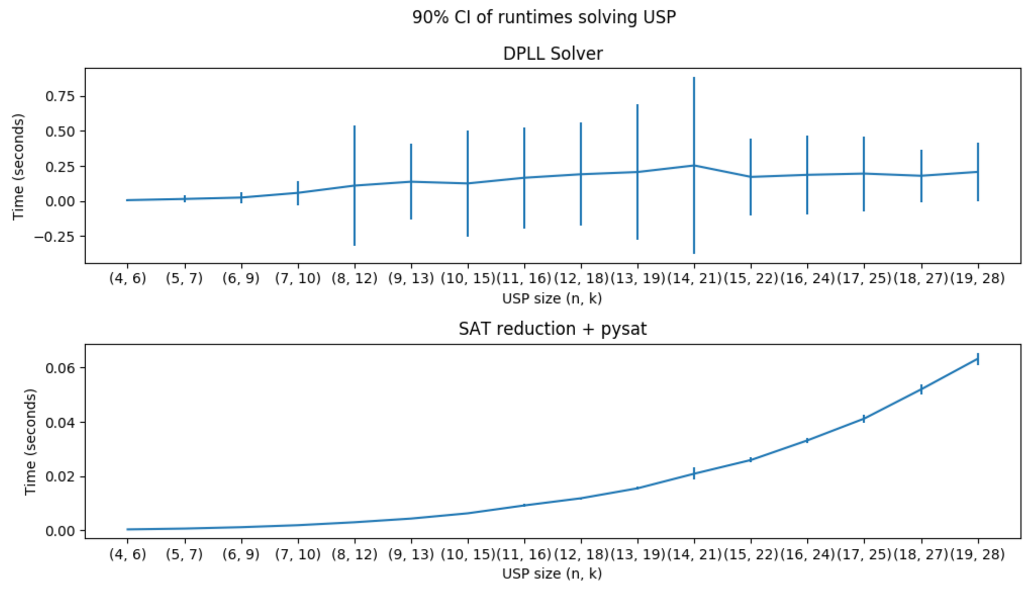
1. a. My DPLL-style solution is included in `USP.py`. Running that file as a script will run the verifier on 1000 randomly generated 4x4 USPs.

   b. Random instances are generated by `GenerateUsp` in `USP.py`. This generator follows the guidelines on random instances for USP in the writeup, namely given n, k, it fills in each $U[i][j]$ independently with $1, 2$, or 3 with equal probability.

   I emperically found square USPs to be strong around 5% of the time. I also found Tall USPs would be strong with much lower probability, while wide USPs would be strong with higher probability. Emperically, it seems choosing dimensions n, k where $k \approx 1.5 * n$ seems to produce a near-equal distribution of strong and weak USPs for relatively small n.

   c. For each data point I generated 100 random USPs and graphed the 90% CI of the runtime. The generated USPs were about 1.5 times wider than tall, since I found that created a pretty equal ratio of weak to strong USPs. It's clear that my DPLL solution has really high variance in it's runtime, since many of the error bars include negative time values. It's somewhat difficult to see the general trend of the line, with how large the error bars are, but it appears to be steadily increasing. The larger USPs take about a quarter of a second on average.



2. Since `USP Weakness` asks us to name valid permutations ρ and σ, the reduction to `CNF-SAT` needs to encode those valid permutations in any satisfying assignment. This leads us to two main steps in the reduction. First, we must create a set of clauses that defines what a valid permutation is, and second, we need to encode the specific USP in another set of clauses.

   We will define two sets of variables x and y, to encode the permutations ρ and σ respectively. These variables have subscripts, such as x_{ij} which if set to true means that the number i is assigned to the j^{th} spot of ρ. As there are n choices for both i and j, we are requiring a total of $2n^2$ variables for a USP of size (n, k). From this definition, we can create clauses which define a valid permutation.

   First, we encode the fact that only one number is assigned in a spot.

$$x_{ij} \implies \{\bar{x}_{kj} \quad | \quad k \neq i\}$$
$$\equiv \bar{x}_{ij} \vee \{\bar{x}_{kj} \quad | \quad k \neq i\}$$
$$\equiv \forall k \neq i : \bar{x}_{ij} \vee \bar{x}_{kj}$$

   Therefore, we can see this statement can be encoded in $\binom{n}{2}$ `CNF-SAT` clauses for each i given an input USP of size (n, k). We repeat this n times, giving a total $\mathcal{O}(n^3)$ clauses to encode this condition for both permutations.

   Next, we want to encode the fact that a number appears only once in the permutation.

$$x_{ij} \implies \{\bar{x}_{ik} \quad | \quad k \neq j\}$$
$$\equiv \bar{x}_{ij} \vee \{\bar{x}_{ik} \quad | \quad k \neq j\}$$
$$\equiv \forall k \neq j : \bar{x}_{ij} \vee \bar{x}_{ik}$$

   Just as before, this statement can be encoded in $\binom{n}{2}$ `CNF-SAT` clauses for each i given an input USP of size (n, k). This is repeated n times, which gives us a total $\mathcal{O}(n^3)$ clauses to encode this condition for both permutations.

   The two above conditions are sufficent to define valid permutations in `CNF-SAT`. However, `USP Weakness` also requires at least one of ρ or σ to not be the identity. Since this is only one possible assignment to ρ and σ, we can simply add one clause to prevent that assignment.

Overall, the time complexity of creating clauses which define valid permutations for an input USP of size (n, k) is $\mathcal{O}(n^3)$, so we see this first step runs in polynomial time to the input size. Now we can move on to the second part, encoding the USP conditions in another set of clauses.

The problem of `USP Weakness` is to find permutations such that the number of satisfied conditions is not two. Before any assignment to any variable, we already know which i, j already have a satisfied condition, namely $U[i][j] = 1$. In this case, we need to assign $\rho(i)$ and $\sigma(i)$ such that $U[\rho(i)][j] = 2 \iff U[\sigma(i)][j] = 3$, which ensures that we will either end up with 1 or 3 satisfied conditions.

The following clauses prevent assignments to ρ and σ for i, j which violate this condition.

$$\forall k, l \text{ s.t. } U[k][j] \neq 2 \ \& \ U[l][j] = 3$$
$$\bar{x}_{ki} \vee \bar{y}_{li}$$

$$\forall k, l \text{ s.t. } U[k][j] = 2 \ \& \ U[l][j] \neq 3$$
$$\bar{x}_{ki} \vee \bar{y}_{li}$$

Generating these clauses requires looping through the entire input USP, and then checking the j^{th} element of each row. This step therefore takes $\mathcal{O}(kn^2)$ time.

Finally, we need to consider i, j where $U[i][j] \neq 1$. In this case, we can't have both $U[\rho(i)][j] = 2$ and $U[\sigma(i)][j] = 3$, as that would lead to two satisfied conditions.

The following clauses prevent assignments to ρ and σ for i, j which violate this condition.

$$\forall k, l \text{ s.t. } U[k][j] = 2 \ \& \ U[l][j] = 3$$
$$\bar{x}_{ki} \vee \bar{y}_{li}$$

As above, generating these clauses requires looping through the input USP, and then checking the j^{th} element of each row. This step takes $\mathcal{O}(kn^2)$ time.

At this point, we have removed every possible assignment to ρ and σ which would lead to two conditions holding for all i, j. This, combined with the clauses encoding semantics for valid permutations is sufficent to reduce `USP Weakness` to `CNF-SAT`. Each individual step of that reduction happened in polynomial time to the input to `USP Weakness`, therefore this is a polynomial time reduction to `CNF-SAT`.

3. The graph for the runtime of my SAT-reduction and pysat algorithm is included in the same figure as 1.c. I ran this algorithm on the same generated USPs, meaning any weird USP that ran slowly on my DPLL solution was tested on the SAT-reduction plus pysat solution.

   The code for my reduction is in `Sat.py`, running that file as a script will verify the algorithm's correctness on 1000 randomly generated USPs of size (20, 20). The code to generate the graphs is in `Graphs.py`.

   As far as general trends, it's clear that the SAT-reduction route produced a much more consistent algorithm. Unfortunately, the confidence interval for my DPLL solution is too large to make any definitive statement on which algorithm performed better, but the means suggest the SAT reduction + pysat was more efficient as they took less than $\frac{1}{4}$ of the time on average.