Sat solver:
Our solution loads its sat solver from folder the folder ../satsolver relative to ex3.pl according to the structure of our workspaces

General flow of sum_equals(Sum, Numbers, CNF):

- Recursively add the binary numbers in Numbers - the first two numbers are replaced by the sum of their addition which is also binary representation composed of unknown variables
- Each two numbers are added using fullAdder component where the carry in of the first one is 0 (represented as -1) and the carry out of the last one is the MSbit of the sum
- Of course the numbers are not physically summed. These addition operations create a CNF that will satisfy if and only if the sum of the numbers, which are at the moment unknow, will be the given sum
- The resulted sum which is bit variables that are chained to the addition operation is assigned with the actual given sum.
- the our implementation relies on the code and explanations shown in class

main milestones, predicates not necessarily shown in the sum_equals predicate

General flow of kakuroEncode(Instance+, Map-, CNF-):

- Map all variables
- For each variables under the form: Sum = Variables generate the correctness kakuro CNF using sum_equals and all_diff from previous tasks
- Force all variables to be between 1 and 9 according to kakuro's rules (and the instructions) with two additional CNF expressions.

main milestones, predicates not necessarily shown in the kakuroEncode predicate

A note regarding kakuroDecode:
While implementing the encoding predicate we designed our Map to be as intuitive and readable as possible(for us at least). We designed map to be in the form of `[... | Var_i = [B_i0, B_i1, B_i2, B_i3] | ....]` where each variable appears only once and is identical to the ones we get in instance. This design has proved itself to be a good pick since we had to uncover and fix many bugs that were exposed when we merged our implementation. The general Idea was that Solution would be in fact identical to Instance where the assignment for the given variables will fill it with the correct solution to the kakuro board. When we stepped forward to implement kakuroDecode we realized that we have no way of creating the Solution within the predicate basing only on the above original design. We modified the Map structure to be in the form of [Instance , VarsMapping], where VarsMapping is as described above.
For example, for the following instance: [3 = [V1, V2]] the kakuroEncode will create the Map: [[3 = [V1, V2]], [V1 = [V1_0, V1_1, V1_2, V1_3], V2 = [V2_0, V2_1, V2_2, V2_3]]]

The generated CNF will contain all Vi_k, the sat solver will provide the assignment for them and the decode will convert every Vi_k bit into Vi decimal number for the solution.

General flow of kakuroSolve(Instance+, Solution-):

- Encode the instance into Map and CNF
- Apply the sat solver for the CNF
- Decode Map which will set values for variables mentioned in Instance, making Solution whole
- Validate the solution

Tests:

As mentioned earlier, it was highly difficult to trace and fix the bugs, prior and after merging our implementations, while not breaking something else pushed us to implement some tests. These tests also helped us run again and again problematic test cases until we managed to fix them.

We decided to add the tests file tdd.pl for you comfort. Simply load the file using the command “[tdd].” from prolog terminal opened in the same directory is both of the files(ex3.pl, tdd.pl).

We, Gal Tfilin (303097109) and Yarden Chen (305509069), assert that the work we submitted is entirely or own. We have not received any part from any other student in the class (or other source), nor did we give parts of it for use to others. We have clearly marked in the comments of our program any code taken from an external source. We assert that the division of work between us is: Gal 50% and Yarden 50%.