Techniques for Improving Software Productivity
Ex. 1: SAT and SMT
Due 31/03/2019

Submission Instructions
Please read, and follow, the submission instructions.

Code
The code skeleton for the exercise can be found in: https://bitbucket.org/tausigplan/soft-prod19 under exercises/ex1/, and the code from the demos can be found there under demos/.

SAT (40%) – k-clique (`k_clique.py`)

A *clique* of size k in a graph $G = (V, E)$ is a fully connected subgraph of G with k vertices, i.e., $G' = (V', E')$ s.t. $V' \subseteq V$, $E' \subseteq E$, $|V'| = k$, and $\forall v, u \in V'. (v, u) \in E'$.
In this question you are requested to implement a reduction from the k-clique problem (finding a clique of size k) to SAT and apply Z3 to obtain a solution.

Implement the function `get_k_clique(k, V, E)`, under the following assumptions:

- `k` is an integer value ($k > 0$).
- `V` is the list `[0,1,...,n-1]` where `n` is the number of vertices.
- `E` is a list of edges, where each edge is represented by a pair (2-tuple) of vertices.
- The function return value should be `None` if there is no k-clique in the given graph, or a list of vertices if there is a k-clique.

You may use the function `draw_graph` to visualize your results.

Unsat Core (40%) – k-vertex-cover (`k_vertex_cover_core.py`)

A *vertex cover* of size k in a graph $G = (V, E)$ is a set of vertices of size k such that each edge in the graph is connected to at least one vertex in the set. i.e., $S \subseteq V$ s.t. $|S| = k$, $\forall e \in E. \exists v \in S, u \in V. e = (v, u) \vee e = (u, v)$.
In this question you are requested to implement a reduction from the k-vertex-cover problem (finding a vertex cover of size k) to SAT, apply Z3 to obtain a solution in case a solution exists, and r a subgraph $G' = (V, E')$ in case it does not, s.t. $E' \subseteq E$, and there is no k-vertex-cover of G' as well.

Implement the function `get_k_vertex_cover(k, V, E)` under the following assumptions:

- `k` is an integer value ($k > 0$).
- `V` is the list `[0,1,...,n-1]` where n is the number of vertices.
- `E` is a list of edges, where each edge is represented by a pair (2-tuple) of vertices.
- If there is a k-vertex-cover of the given graph, the function returns a list of vertices.

- If there is no k-vertex-cover, the function uses the unsatisfiable core mechanism of Z3 to obtain a set of edges $E' \subseteq E$, such that the graph $G' = (V, E')$ also does not have a k-vertex-cover. The return value in this case should be a list of edges (i.e., a list of 2-tuples).

You may use the function `draw_graph` to visualize your results.

Note: Simply returning the set E, without obtaining an unsat-core, is not a valid solution. While we encourage the use of formal logic in your reasoning, we will insist on following the spirit of the law in this case.
In some cases, the unsat-core mechanism of Z3 will return the full problem. That is ok.

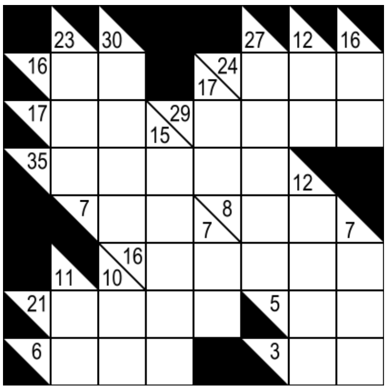
SMT (40%) – Kakuro (`kakuro.py`)

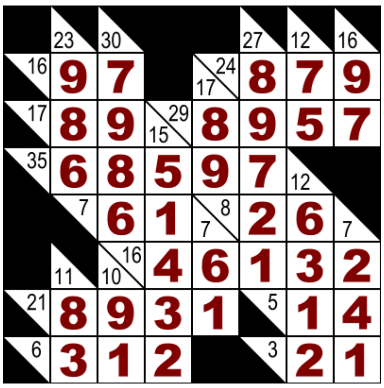
Kakuro is a mathematics puzzle similar to Sudoku.

From the Wikipedia article on Kakuro:

> The canonical Kakuro puzzle is played in a grid of filled and barred cells, "black" and "white" respectively […] Apart from the top row and leftmost column which are entirely black, the grid is divided into "entries"—lines of white cells—by the black cells. The black cells contain a diagonal slash from upper-left to lower-right and a number in one or both halves, such that each horizontal entry has a number in the black half-cell to its immediate left and each vertical entry has a number in the black half-cell immediately above it. These numbers, borrowing crossword terminology, are commonly called "clues".
>
> The objective of the puzzle is to insert a digit from 1 to 9 inclusive into each white cell such that the sum of the numbers in each entry matches the clue associated with it and that no digit is duplicated in any entry. It is that lack of duplication that makes creating Kakuro puzzles with unique solutions possible, and which means solving a Kakuro puzzle involves investigating combinations more, compared to Sudoku in which the focus is on permutations. There is an unwritten rule for making Kakuro puzzles that each clue must have at least two numbers that add up to it, since including only one number is mathematically trivial when solving Kakuro puzzles.

Example of a Kakuro board, and its solution:





In this question you will reduce the Kakuro problem to satisfiability modulo theory (SMT) and use Z3 to solve it.

Implement the function `get_kakuro_solution(board, nrows, ncols)` under the following assumptions:

- `board` is an `nrows` x `ncols` matrix (list of list). Cells may be:
  - The string `‘B’` for an empty blacked out cell
  - The string `‘W’` for an empty white cell
  - A pair (2-tuple), that represents a clue or a pair of clues. The first element of the tuple is the clue in the downwards direction. The second element is the clue in the rightwards direction. One of the cells may be `‘B’`.
    In the example above, the second cell in the first row will be `(23,’B’)`. The fifth cell on the second row will be `(17,24)`
  - **Note:** The parameter `board` is a **list of rows**. Each **row** is a **list of cells**. That is, `board[2]` is the third row of the board and `Board[2][3]` is the cell in the third row and the fourth column.
    In the example above we will have that `board[2]` is:
    `[(‘B’,17), ’W’ ,’W’,(15,29) ,’W’ ,’W’ ,’W’ ,’W’]`
    For further information, consult the input-output examples in the file.
- `Nrows` is the number of rows in the board (including the topmost row). In the example above, there are **8** rows.
- `Ncols` is the number of columns in the board (including the leftmost column). In the example above, there are **8** columns.
- The function returns a `nrows` x `ncols` matrix, which is identical to the input matrix, but with the solution instead of the cells that held the value `‘W’` in the input
  For further information consult the input-output examples in the file
- If no solution exists, the function should return None

You may use the function `draw_board` to visualize your results.

Hint: Z3 provides the useful function `Sum`