Automatic Software Verification
Ex. 1: SAT and SMT
Due 21/04/2015

Code
The code skeleton for the exercise can be found in: https://bitbucket.org/tausigplan/asv15 under exercises/ex1/, and the code from the demos can be found there under demos/.

1. k-edge-coloring (k_edge_coloring.py)

An edge coloring of a graph is an assignment of colors to the edges of the graph so that no two adjacent edges have the same color (edges are adjacent if they share a common vertex). In this question you are requested to implement a reduction from the k-edge-coloring problem (edge coloring with k colors) to SAT, and apply Z3 to obtain solutions, and also an unsatisfiable core.

a. Implement the function get_k_edge_coloring(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-edge-coloring of the given graph, or a dictionary mapping edges (represented as 2-tuples) to colors (represented by numbers between 0 and k-1) if there is a k-edge-coloring.

b. Implement the function get_k_edge_coloring_core(k, V, E) under the following assumptions:
   - arguments are the same as get_k_edge_coloring.
   - If there is a k-edge-coloring of the given graph, output is the same as in get_k_edge_coloring.
   - If there is no k-edge-coloring, the function uses the unsatisfiable core mechanism of Z3, to obtain a set of edges E', such that the graph (V, E') also does not have a k-edge-coloring. The return value in this case should be a dictionary mapping edges in E' to 1 (with edges not in E' unmapped).

You may use the function draw_graph to visualize your results.

Test your implementation on the Petersen graph and make sure you get a 4-edge-coloring but not a 3-edge-coloring (the Petersen graph is not 3-edge-colorable).

Note: Your implementation must be correct on other graphs as well - test it as much as you think needed, and not only on the Petersen graph.

2. Transport planning (planning.py)

We define the following transport planning problem:

There are nc cities, np packages and na airplanes. Every package has a source city and a destination city, and every airplane has a starting city. Transport of the packages is done in steps, where at each step each airplane can move between cities, or stay in the same city and load or unload packages. The airplanes move independently, and if at a step an airplane A does not move - multiple packages can be loaded or unloaded from A. There is no limit to the number of packages that a single airplane can contain.

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

To formalize the problem using Z3, define the following uninterpreted sorts:

- C for cities
- P for packages
- A for airplanes

And the following predicates and functions:

- at(package, city, time) - a predicate indicating the package is at city during time
- on(package, airplane, time) - a predicate indicating the package is on airplane during time
- loc(airplane, time) - a function that denotes the city where the airplane is located during time

Times in the transport plan are to be represented using integers, with time 0 indicating the initial state, and time j indicating the state after j steps.

A valid plan meets the following constraints:

- At any time, a package can either be on some airplane or at some city (but not both).
- An airplane can load and unload packages to the city where it is located, but only in steps where it does not move.
- Further constraints - formalize them from the problem definition.

Implement the function get_transport_plan(nc, np, na, src, dst, start). The function arguments are:

- nc, np, and na are the number of cities, packages and airplanes respectively. Cities are numbered 0 to nc-1, and similarly for packages and airplanes.
- src and dst are lists of length np, with elements in the range 0 to nc-1, representing the source and destination cities of the packages.
- start is a list of length na, with elements in the range 0 to nc-1, representing the start city of the airplanes.

For example, the following input:

nc=4, np=3, na=2, src=[2,1,0], dst=[0,3,2], start=[3,3]

represents the following problem:

- There are 4 cities (call them C0,C1,C2,C3), 3 packages (call them P0,P1,P2) and 2 airplanes (call them A0,A1).
- P0 starts at C2 and should be moved to C0.
- P1 starts at C1 and should be moved to C3.
- P2 starts at C0 and should be moved to C2.
- Both airplanes start at C3.

You may use the given function print_problem to understand the arguments format.

The function get_transport_plan should find the shortest transport plan (minimal number of steps) and return a tuple (city_packages, city_airplanes, airplane_packages), where:

- city_packages is a list of lists of lists. The first dimension represents time, and if the transport plan has k steps, then the length of the first dimension is k+1. The second dimension represents cities, and its length is nc. For every time t and city i, city_packages[t][i] is a list of packages which are at city i during time t. Packages are represented by numbers between 0 and np-1.
- city_airplanes is a list of lists of lists, similar to city_packages. For every time t and city i, city_airplanes[t][i] is a list of numbers between 0 and na-1 representing the airplanes located in city i at time t.
- airplane_packages is a list of lists of lists, where the first dimension represents time, and the second dimension represents airplanes. The length of the second dimension is na. For every time t and airplane i, city_packages[t][i] is a list of packages which are on airplane i during time t.

You may use the function print_plan to understand the requested result format and print the output of your implementation, and also look at example_solution found in planning.py.

Bonus

Improve your planning.py implementatin to find a solution with minimal number of airplane moves (while maintaining minimal number of steps).