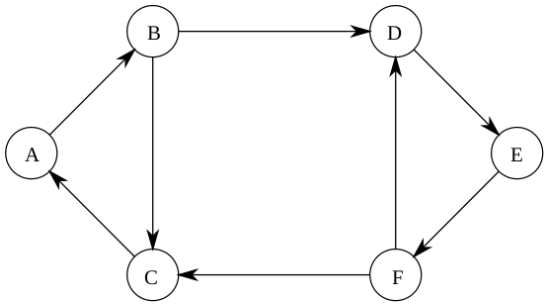
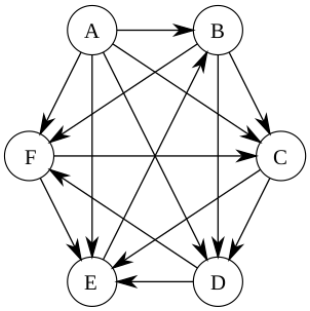
Inspectable Incrementality in Minimum Feedback Arc Set Solving

Colin Shea-Blymyer
sheablyc@oregonstate.edu
Oregon State University
Corvallis, Oregon, USA

Jeffrey M. Young
youngjef@oregonstate.edu
Oregon State University
Corvallis, Oregon, USA



(a) A three cycle graph which demonstrates that greedily selecting the minimum weighted edge will not provide the minimum feedback arc set.



(b) A four cycle graph where each cycle depends on the edge (E, B).

ABSTRACT

The minimum feedback arc set problem is the problem of finding the least amount of edges in a directed, cyclic, graph, such that if these edges are removed a directed, acyclic graph results. There are exists several classic and approximate methods to solve the minimum feedback arc set problem. In this study, we explore the novel use of satisfiability-modulo theories (SMT) solvers to solve the minimum feedback arc set problem. With an SMT solver, we hypothesize that not only is an effective encoding possible, but by employing advanced features of SMT solvers, such as incrementality and unsatisfiable cores, we can provide users with decision points to give users the ability to pause the solution routine, resample particular edges, alter the problem or inspect intermediate results. Our results are negative. We find an effective SMT encoding for the minimum feedback arc set problem but the encoding produces SMT problems that in practice are unreasonably slow. Similarly, we find that to make effective use of incrementality and unsatisfiable cores, one would require information which is itself an instance of the minimum feedback arc set, and thus these features are ill suited to this problem domain.

KEYWORDS

incremental SAT solving, minimum feedback arc set, SMT solving

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Colin Shea-Blymyer and Jeffrey M. Young. 2021. Inspectable Incrementality in Minimum Feedback Arc Set Solving. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The minimum feedback arc set problem is a canonical NP-Complete problem given by Karp [7]. Worse still, Kann [6] showed that the problem is APX-hard. Despite these results finding the minimum feedback arc set of a directed graph is desirable for many domains: such as certain rank-choice voting systems, tournament ranking systems, and dependency graphs in general.

colin citation?

Solutions to the minimum feedback arc set problem are commonly implemented in widely used graph libraries yet all suffer from a distinct flaw. While each implementation provides a solution, the implementations only do so without allowing the user to inspect intermediate steps; which might contain useful domain information. For example, the graph Figure 1b displays a directed graph which encodes a single win tournament system. Each vertex is a player and each edge encodes a win over a contestant, for example we see that player E beat contestant B. Observe that the Figure 1b contains cycles which implies that there is not a linear ordering amongst the contestants of the tournament, and so any ranking of the contestants would violate a win of one contestant over another. Removing the minimum feedback arc set would remove the cycles, which yields a linear order in the tournament that violates the fewest number of wins. In this example, intermediate results are edges which compose the minimum feedback arc set, thus if one had access to the intermediate results one might choose to rematch the opponents rather than remove the win. Crucially, the result of the rematch may change the final minimum feedback arc set. Such a

procedure could therefore increase the confidence in the results of the tournament.

To allow for *inspectable incrementality* we propose a novel direction in solving the minimum feedback arc set problem based on recent advances in satisfiability solving (SAT) and SMT solving. Our approach is to utilize an SMT solver to detect cycles and minimize the weight of the feedback arc set. Incrementality in this approach is given through use of an *incremental* SMT solver and generation of *unsatisfiable cores*. A minimum unsatisfiable core is the minimum set of clauses in a SAT or SMT formula which prevent a SAT or SMT solver run from finding a satisfiable assignment. An incremental solver provides the user the ability to add or remove constraints and thereby direct the solver during runtime. Incrementality in our approach is crucial as it allows the user decision points to interact with the solution process. Thus, a user might observe a unsatisfiable core which corresponds to a cycle and deliberately resample *only* the edges in the discovered cycle.

Unfortunately, we find that incremental SMT for finding a minimum feedback arc set suffers from significant issues which seem to be intractable. In addition to this result we make the following contributions:

(1) An identification of the most intractable problems for an incremental SMT encoding on the minimum feedback arc set problem. (Section 3)
(2) An SMT encoding to find the minimum feedback arc set of an arbitrary directed and cyclic graph. (Section 4)
(3) An empirical evaluation of the SMT encoding. (Section 6)

2 BACKGROUND AND RELATED WORK

In this section, we provide background on the minimum feedback arc set problem, incremental SAT and SMT solving, and unsatisfiable cores. We begin with preliminaries on graphs and feedback arc sets. We close the section with a description of incremental SAT solving and SMT solving.

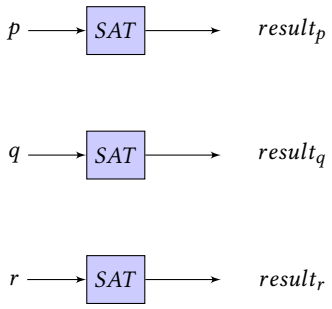
On a directed graph $G(V, E)$ a *feedback arc set* S is a set of edges in E, such that removing them from the graph G results in an acyclic graph. The minimum feedback arc set problem finds the S^* of the minimum size. For weighted graphs, one might desire S^* to have the minimum total weight of any S. Finding this S^* is the minimum weighted feedback arc set problem.

With the minimum feedback arc set problem defined, we provide the following background on incremental SAT solvers and assume knowledge of SMT solvers. Suppose, we have three related propositional formulas that we desire to solve.

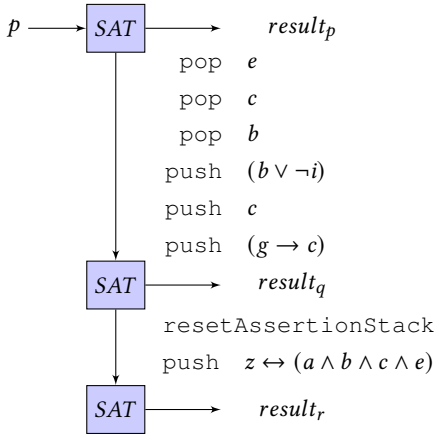
$$p = a \wedge b \wedge c \wedge e$$
$$q = a \wedge (b \vee \neg i) \wedge c \wedge (g \rightarrow c)$$
$$r = z \leftrightarrow (a \wedge b \wedge c \wedge e)$$

p is simply a conjunction of variables. In q, relative to p, we see that the variables i and g are added, e is removed, and there are two new clauses: $(b \vee \neg i)$ and $(g \rightarrow c)$, both of which possibly affect the values of b and c. In r, the variables and constraints introduced in p are further constrained to a new variable, z.

Suppose one wants to find a model for each formula. Using a non-incremental SAT solver results in the procedure illustrated in Figure 2a; where SAT solving is a batch process and no information is reused. Alternatively, a procedure using an incremental SAT solver is illustrated in Figure 2b; in this scenario, all formulas are solved by single solver instance where terms are programmatically added or removed from the solver.



(a) Brute force procedure, no reuse between solver calls.



(b) Incremental procedure, reuse defined by POP and PUSH.

Figure 2: Comparison of incremental and non-incremental SAT procedures.

The ability to add and remove terms is enabled by manipulating the *assertion stack*, to add or remove levels on the stack. The incremental interface provides two commands: PUSH to create a new *scope* and add a level to the stack, and POP to remove the topmost level on the stack. Consider the following SMTLIB2 program which demonstrates three levels on the assertion stack. The SMTLIB2 standard [2] is an international standard that defines a general interface for SAT and SMT solvers. The program follows the procedure outlined in Figure 2b and solves p, q and r:

```
(declare-const a Bool)          ;; variable declarations for p
(declare-const b Bool)
(declare-const c Bool)
(declare-const e Bool)
(assert a)                      ;; a is shared between p and q
(push)                          ;; solve p
  (assert e)
  (assert c)
  (assert b)
  (check-sat)                   ;; check-sat on p
(pop)                           ;; remove e, c, and b assertions
```

```
(push)                          ;; solve for q
  (declare-const i Bool)        ;; new variables
  (declare-const g Bool)
  (assert (or b (not i)))       ;; new clause
  (assert c)                    ;; re-add c
  (assert (=> g c))             ;; new clause
  (check-sat)                   ;; check sat of q
(pop)                           ;; i and g out of scope
(reset)                         ;; reset the assertion stack
(declare-const a Bool)          ;; variable declarations for r
(declare-const b Bool)
(declare-const c Bool)
(declare-const e Bool)
(declare-const z Bool)
(assert (= z (and a (and b (and c (and e ))))))
(check-sat)                     ;; check-sat on r
```

We begin by defining p, and assert a outside of a new scope so that it can be reused for q. Internally, all levels on the assertions stack are conjoined and asserted when a CHECK-SAT command is issued. Thus, we reuse a by exploiting this conjunction behavior. Had we asserted (and a (and b (and c (and e)))), then we would not be able to reuse only the assertion on a since it was created in conjunction with other variables. The first PUSH command enters a new level on the assertion stack, the remaining variables are asserted and we issue a CHECK-SAT call. After the POP command, all assertions and declarations from the previous level are removed. Thus, after we solve q the variables i and g cannot be referenced as they are no longer in scope. Similarly, after the first CHECK-SAT call and subsequent POP, e, c and b are no longer defined.

In an efficient process one would initially add as many *shared* terms as possible, such as a from p and then reuse that term as many times as needed. An efficient process should perform only enough manipulation of the assertion stack as required to reach the next SAT problem of interest from the current one. However, notice that doing so is not entirely straight forward; we were only able to reuse a from p in q because we defined p in a non-intuitive way by utilizing the internal behavior of the assertion stack. This situation is exacerbated by SAT problems such as r, where due to the equivalence between a new term and shared terms, we are forced to completely remove everything on the stack with a RESET command just to construct r. Thus incremental SAT solvers provide the primitive operations required to solve related SAT problems efficiently, yet writing the SMTLIB2 program to solve the set efficiently is not straightforward.

3 PROBLEMS WITH AN INCREMENTAL ENCODING

Our original conception for this study was to explore the use of incremental SAT and SMT solvers to solve the minimum feedback arc set problem. However, we found fundamental problems with this approach during several courses of experimentation. In this section, we review the problematic nature of the incremental approach thereby enriching the problem. We provide a working non-incremental SMT encoding in the next section.

The incremental approach was a combination of two SAT and SMT features. First, we desired to encode the problem in such a way that an UNSAT would be returned from the solver. With an UNSAT returned, we could then query for an *unsatisfiable core*, that is, the set of constraints which prevent the solver from unifying. Second, we sought to use *scopes* to add and remove clauses from the incremental solver via PUSH and POP calls. With scopes it is possible to refine the constraints *during* the solving routine, possibly simplifying the problem space. We hypothesized that by using scopes and unsatisfiable cores we could reveal domain information to the user and direct the solver to a solution in an efficient manner.

Unfortunately the incremental approach has numerous problems for finding the minimum feedback arc set, which appear to be intractable. First, in order to generate unsatisfiable cores constraints must be added to the solver instance *and watched*. In and of itself, this is not an intractable problem, the SMTLIB2 standard defines a function to track constraints called ASSERT_AND_TRACK which takes a constraint and a name. The name is then returned in the unsatisfiable core if the concomitant constraint is in the core. The issue becomes a constant time cost incurred by parsing the unsatisfiable core into a usable format for the solving routine to continue. The pattern becomes: query for an unsatisfiable core, parse the core, refine the constraints, and then repeat. Thus the constant time cost is exacerbated for every core that is generated. In addition, there is no guarantee that the unsatisfiable core itself is minimal, although some SAT/SMT solvers, such as z3 [4] expose a setting to produce minimal cores at the expense of performance.

The more serious issue is the use of scopes is problematic for tackling the minimum feedback arc set problem at all. While scopes would allow one to direct the solver to a solution, or explore possible solutions, it requires knowledge of the problem domain which is not readily available during the encoding process. In order to employ scopes, one must know before the encoding step which constraints are likely to change. This knowledge is required so that the constraints can be pushed onto the assertion stack *as late as possible*. When these constraints are on top of the assertion stack (and thus pushed last) they can be removed from the assertion stack *without* removing constraints that we do not wish to refine. Without this knowledge, there is a substantial risk that the we may need to remove all, or almost all constraints from the solver just to repack the solver and refine the query. In the worst case, this requires an additional complete traversal of the graph. The issue is further exacerbated by the minimum feedback arc set problem because it is not possible to know before hand which edges, and thus which constraints, will need to be refined. In fact finding the unique set of constraints that will need to be refined is itself an instance of finding the minimum feedback arc set!

4 APPROACH AND SMT ENCODING

Our approach is to translate an exact integer programming method by Baharev et al. [1] to an SMT problem. The translation to an SMT problem is straightforward, requiring no changes from linear program encoding to an SMT encoding.

To find the minimum feedback arc set of a graph G, the method requires a cycle matrix, a_{ij}. The cycle matrix tracks which edges are in which cycles in the graph. For some edge e_j, if e_j participates in cycle i then $a_{ij} = 1$ otherwise $a_{ij} = 0$, indicating that e_j *does not* participate in the ith cycle. For example, consider the edge (B, D) from Figure 1a and assume (B, D) has edge ID $j = 3$, if the 0th cycle in Figure 1a is composed of edges: (A, B), (B, C), and (C, A), then

Edge	ID
(A, B)	0
(B, C)	1
(C, A)	2
(B, D)	3
(D, E)	4
(E, F)	5
(F, C)	6
(F, D)	7

Cycle	Cycle edges
0	(A, B) → (B, C) → (C, A)
1	(D, E) → (E, F) → (F, D)
2	(A, B) → (B, D) → (D, E) → (E, F) → (F, C) → (C, A)

Table 1: Table of Edge IDs and Cycle IDs for Figure 1a

$a_{03} = 0$. Similarly, assuming (A, B) has edge ID $j = 0$, then $a_{00} = 1$ because (A, B) participates in cycle 0.

With the cycle matrix, the method is composed of c constraints, where c is the number of cycles and a minimization over the sum of edges. Each edge, (i, j) is encoded as a binary integer SMT variable e_j, where j is the edge's unique ID. The SMT variable e_j only ranges from 0 to 1, indicating whether the jth edge is in the minimum feedback arc set ($e_j = 1$) or not ($e_j = 0$). We express the constraints in the following equations:

$$\text{minimize} \sum_{j=1}^{|E|} e_j \tag{1}$$

$$(\sum_{j=1}^{|E|} a_{ij} e_j) \geq 1, \text{ for each } i = 0, 1, 2 \ldots, c \tag{2}$$

Conceptually, Equation 1 requires the SMT solver to consider all edges in the graph G and find the smallest number of edges which compose a feedback arc set. The encoding of a feedback arc set occurs in Equation 2. With the constraint $\sum(\ldots) \geq 1$ for the ith cycle, Equation 2 states that for each cycle in the graph G pick one or more edges. By $\sum(a_{ij}e_j) \geq 1$, Equation 2 forces the SMT solver to consider edges that occur in most cycles. The logic is easiest to observe with an example, consider the graph shown in Figure 1a, with the edge and cycle IDs shown in Table 1: For this graph, the constraints generated by Equation 2 would be:

$1e_0 + 1e_1 + 1e_2 + 0e_3 + 0e_4 + 0e_5 + 0e_6 + 0e_7 \geq 1$ Cycle 0

$0e_0 + 0e_1 + 0e_2 + 0e_3 + 1e_4 + 1e_5 + 0e_6 + 1e_7 \geq 1$ Cycle 1

$1e_0 + 0e_1 + 1e_2 + 1e_3 + 1e_4 + 1e_5 + 1e_6 + 0e_7 \geq 1$ Cycle 2

Where we see the edge e_0 (edge (A, B)) occurs in cycles 0, and 2 (as a_{00} and $a_{20} = 1$). Therefore, the easiest way for the SMT solver to satisfy these constraints *and* the minimization constraint is to set $e_0 = 1$. However, doing so will not satisfy the cycle constraints for cycle 1, thus the solver must pick either e_4 ((D, E)), e_5 ((E, F)), or e_7 ((F, C)). Again the simplest path is to pick e_4, or e_5 as setting these variables to 1 satisfies more than one constraint. Thus, we see that by this encoding the SMT solver is forced to minimize the number of edges to pick, and therefore maximize the number of cycle constraints satisfied by setting a given edge to 1, which corresponds to finding the minimum feedback arc set of a graph.

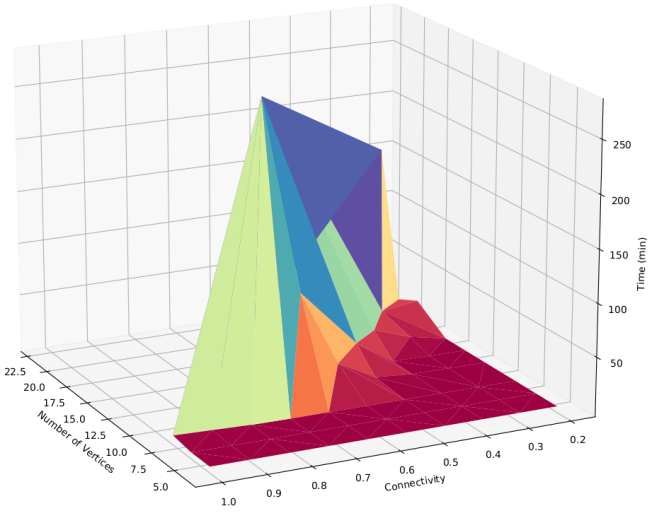


Figure 3: Results of the SMT encoding on randomly generated Erdős-Rényi graphs. The z-axis is runtime in minutes, with connectivity on the y-axis and vertex count on the x-axis. We see that runtime explodes at an interaction point between vertex count and connectivity.

5 CASE STUDIES

To evaluate this approach, we construct a prototype SMT-enabled algorithm and assess the prototype on Erdős-Rényi graphs. Erdős-Rényi graphs are generated according to two parameters: p, a metric of connectedness for the graph, and k is the number of vertices in the graph.

With these parameters we employ random generation to construct sample graphs. We are interested in the individual effect k and p have on runtime, in addition to the interactions effects between each parameter. Consider the case where p and k are left unbound, yet s is set to 1. This is the specific case where solving the minimum weighted feedback arc set problem solves the minimum feedback arc set problem. Thus, by setting s to 1 we generate graphs to solve the minimum feedback arc set problem. Consider the case where s is larger than one. In this case, edges must possess positive weights and the weighted minimum feedback arc set may be different than the minimum feedback arc set.

We provide the complete prototype implementation in Appendix A. To ensure correctness we compared results from the SMT routine to a built in method in the python library igraph which finds the minimum feedback arc set. We observed no differences between both methods.

6 RESULTS AND DISCUSSION

Figure 3 displays the runtime of the prototype SMT method to find a minimum feedback arc set for randomly generated Erdős-Rényi graphs. We observe that as the number of vertices increases the runtime exponentially increases, for example with x = 5 vertices the runtime is stable, however at x = 12 the runtime becomes greater than 2 hours (clock time). Similarly, highly connected graphs the runtime exponentially increases. Consider the difference between

20% connectivity and 70% connectivity, at 20% we see the runtime stay low as vertex count increases, however at 70% connectivity the runtime quickly grows to the order of hours for a graph with just 10 vertices. Lastly, we observe an interaction between the vertex count and connectivity parameters which creates a runtime barrier.

Our observations are not entirely unexpected. Since our method is centered around a cycle matrix the number of cycles in the graph should impact the runtime of the solver. What is unexpected in our study is the magnitude of the runtime performance. Industrial strength SAT and SMT solvers are used to solve constraints with millions of variables and several million clauses [3], yet our results here indicate a ludicrously difficult problem for the SMT solver to solve.

We speculate on possible causes. First, it could be that our prototype is translating the constraints to its internal z3 instance poorly, leading to a high degree of garbage collection pressure in the python runtime. Second, our encoding could accidentally be creating SMT problems that are in the *phase change* [5] region of the SMT solver.

The phase change of SAT and SMT solvers is the region where difficult to solve problems occur. The phase change is characterized by the ratio of clauses to variables. Conceptually if we have many clauses but not many variables then we are over-constrained and thus it is easy to find UNSAT. However, if we have many variables and not many clauses then we are under-constrained and it is easy to find SAT. The difficult region for a SAT or SMT solver occurs when the number of clauses and variables are balanced. Thus, it could be the case that our encoding creates SMT problems in this range. Similarly, it could be that our encoding as a simple summation over symbolic variables is naive. A more efficient implementation would use high performance SMT structures such as BIT-VECTORS [2] which have been successfully used to solve constraint networks in electronic design[3].

Third, there might be specific features of graphs which lead to easier or more difficult SMT problems. For example, previous research has found that tournament graphs, such as Figure 1b are easier to solve than the general case.

REFERENCES

[1] Ali Baharev, Hermann Schichl, Arnold Neumaier, and Tobias Achterberg. 2021. An Exact Method for the Minimum Feedback Arc Set Problem. *Journal of Experimental Algorithmics* 26 (04 2021). https://doi.org/10.1145/3446429

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org.

[3] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. 2009. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, NLD.

[4] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[5] Ian P. Gent and Toby Walsh. 1994. The SAT Phase Transition. In *In Proc. ECAI-94*. 105–109.

[6] Viggo Kann. 1992. On the Approximability of NP-complete Optimization Problems. (01 1992).

[7] Richard M. Karp. 1972. *Reducibility among Combinatorial Problems*. Springer US, Boston, MA, 85–103. https://doi.org/10.1007/978-1-4684-2001-2_9

A APPENDIX

Source code listings

```
"""- Module     : app.py
- Copyright : (c) Jeffrey M. Young
; (c) Colin Shea-Blymyer
- License    : BSD3
- Maintainer: youngjef@oregonstate.edu,
    sheablyc@oregonstate.edu
- Stability  : experimental

"""

from pprint     import pprint
from functools  import reduce
import z3       as z
import igraph   as ig
import numpy    as np

import matplotlib.pyplot as plt
import timeit
from tqdm import tqdm

import graphs  as gs
import utils   as u

def MFAS_set_cover(s,graph):
    """Find the minimum feedback arc set by
        encoding it as a minimum set cover.
    The encoding requires a cycle matrix which we
        find externally to the SAT
    solver. Then given the cycle matrix we do the
        following encoding:

    Variables:
      - m is |E|
      - w_{j} is the weight of edge j \in E (we
          don't implement the weight matrix)
      - y_{j} is a symbolic edge; y_{j} = 1 if
          edge j is in the feedback edge set and 0
           otherwise
      - a is the cycle matrix
      - a_{ij} is the value of edge j in cycle i;
          a_{ij} = 1 if j participates, 0
          otherwise


    minimize(sum{j = 1}^{m}(w_{j} * y_{j}))

      subject to:

    sum_{j = 1}^{m}(a_{ij} * y_{j}) >= 1
    \forall i. y_{i} \in {0,1}
    """

    ## initialization
    m            = graph.ecount()
    cycle_matrix = u.mk_cycle_matrix(u.
        find_all_cycles(graph), m)
    n, c         = graph.get_adjacency().shape
    num_cycles   = len(cycle_matrix)
    edge_list    = graph.get_edgelist()
    sym_to_edge_cache = {}
    edge_to_sym_cache = {}
    sum_var      = 'y'


    def symbolize(i,j):
```

A APPENDIX

Source code listings

```
        "given two indices, create a symbolic
            variable"
        new = z.Int('{0}->{1}'.format(i,j))
        return new


    def constraint_1(i,s_edge):
        """ Multiply the edge by its corresponding
             value in the cycle matrix
        """
        edge  = sym_to_edge_cache[s_edge]
        value = 0
        if edge in cycle_matrix[i]:
            value = cycle_matrix[i][edge]

        return (value * s_edge)


    ## symbolize the edges
    for source,sink in edge_list:
            s_edge                          =
                symbolize(source, sink)
            ## an edge is either a 0 or a 1
            s.add(z.Or([s_edge == 0, s_edge == 1])
                )

            sym_to_edge_cache[s_edge]       = (
                source,sink)
            edge_to_sym_cache[(source,sink)] =
                s_edge


    ## Perform constraint 1 and add it to the
        solver instance
    for i in range(num_cycles):
        s.add(z.Sum([constraint_1(i,s_edge)
                     for s_edge in
                         sym_to_edge_cache.keys()
                         ]) >= 1)


    ## we want the smallest y possible
    s.minimize(z.Sum([s_edge for s_edge in
        sym_to_edge_cache.keys()]))

    s.check()
    return s.model()
```

```
def runWithGraph(graph):
s = z.Optimize()
return MFAS_set_cover(s, graph), u.
    get_feedback_arc_set(graph)


def runErdosRenyi(n,p):
"""Given n vertices and a probability, p of
    edges. Find the minimum
feedback arc set of an erdos-renyi graph

"""
s = z.Optimize()
g = ig.Graph.Erdos_Renyi(n, p, directed=True,
    loops=True)
while g.is_dag():
g = ig.Graph.Erdos_Renyi(n, p, directed=True,
    loops=True)

return MFAS_set_cover(s,g), u.
    get_feedback_arc_set(g)

def runWattsStrogatz(dim, size, nei, p):
"""Given the dimension of the lattice, size of
    the lattice along all
dimensions, the number of steps within which two
     vertices are connected
(nei), and the probability p, find the minimum
    feedback arc set of a
watts-strogatz graph

"""
s = z.Optimize()
g = ig.Graph.Watts_Strogatz(dim, size, nei, p,
    loops=True, multiple=False)
while g.is_dag():
g = ig.Graph.Watts_Strogatz(dim, size, nei, p,
    loops=True, multiple=False)

return MFAS_set_cover(s,g), u.
    get_feedback_arc_set(g)
```

```
"""
- Module     : utils.py
- Copyright  : (c) Jeffrey M. Young
             ; (c) Colin Shea-Blymyer
- License    : BSD3
- Maintainer: youngjef@oregonstate.edu,
    sheablyc@oregonstate.edu
- Stability  : experimental

Common utility functions
"""

from z3 import *
import networkx as nx
from numpy import empty
from re import split

def make_name(frm,to): return frm + "->" + to

def parse_edge(edge): # return list(map(int,edge.
    __str__().split("->")))
    str_edge = edge.__str__()
    inner    = list(map(lambda x: x, str_edge.
        split("->")))
    return tuple(map(lambda x : int(x), inner))

def parse_core(core):
    """Parse an unsat core. An unsat core is
        shallowly embedded as a list of z3
    BoolRef objects such as: [1->2, 2->3, 3->4, 4-
        >1], to operate on these we
    need to coerce them to a string parse the
        string and coerce the Ints out.

    Input:  List of strings, e.g.,      [1->2   , 2
        ->3   , 3->4   , 4->1  ]
    Output: List of List of Ints, e.g., [[1, 2],
        [2, 3], [3, 4], [4, 1]]

    """
    return list(map(lambda e: parse_edge(e), core)
        )

def edge_to_list_dict(g):
    """ Convert a graph to a dictionary of edges
    """
    ret = {}
    for source,sink in g.get_edgelist():
        if source not in ret.keys():
            ret[source] = []

        ret[source].append(sink)

    return ret

def remove_edge(g,source,sink):
    """Given an igraph graph, a source vertex and
        a sink vertex remove the edge
    connecting the source and the sink from the
        igraph graph. This function
    mutates g.
    """
    g.delete_edges(g.get_eid(source,sink))


def flatten(list_o_lists):
    return [e for sublist in list_o_lists for e in
         sublist]


def find_all_cycles(graph):
    nx_graph = graph.to_networkx()
    return list(nx.simple_cycles(nx_graph))


def pairs(ls, n = 1):
    return list(zip(ls, ls[n:] + ls[:n]))


def mk_cycle_matrix(cycle_edge_list, num_edges):
    ps          = [pairs(edges) for edges in
        cycle_edge_list]
    cycle_count = len(cycle_edge_list)
    matrix      = []
    for i in range(cycle_count):
        cycle = {}
        for pair in ps[i]:
            cycle[pair] = 1

        matrix.append(cycle)

    return matrix

def get_feedback_arc_set(graph):
    fas = graph.feedback_arc_set(method="ip")
    return list(map(lambda x : graph.es[x].tuple,
        fas))
```