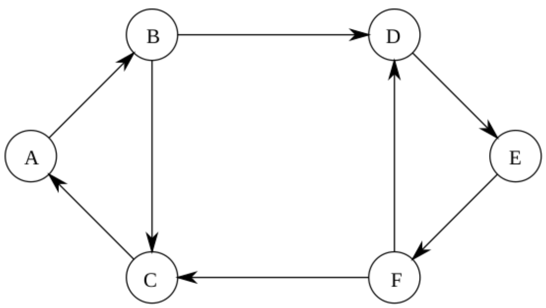
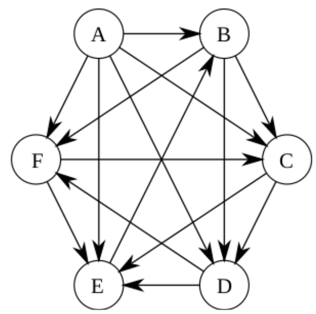
Inspectable Incrementality in Minimum Feedback Arc Set Solving

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

Abstract here

KEYWORDS

datasets, neural networks, gaze detection, text tagging

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, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The minimum feedback arc set problem is a canonical NP-Complete problem given by **?**]. Worse still, the problem is APX-hard yet 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.

add citation

cite Viggo Kann thesis

cite

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 satisfiability-modulo theories (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 end-user the ability to add or remove constraints and thereby direct the solver during runtime. Incrementality in our approach is crucial as it the end-user decision points to interact with the solution process. Thus, an end-user might observe a unsatisfiable core which corresponds to a cycle and deliberately resample *only* the edges in the discovered cycle.

but is it novel?

The approach has benefits ...

We make the following contributions:

(1) Gadgets

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

again, is it novel?

(2) Evaluation of unsatisfiable cores for this problem domain
(3) empirical evaluation of this novel method

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.

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.

Hi Mike! This is a special note environment just for you. From this point we plan on giving an overview of the necessary interface for sat and smt solving. We're planning on adapting this from Jeff's PhD thesis, so it won't be from whole cloth.

3 APPROACH AND SAT ENCODING

Listing 1: Cycle check gadget. `v_from` and `v_to` are symbolic variables in the solver. `name` is the constraint label in the solver. The cycle check replaces each edge with the constraint that previous vertices must be a lesser integer value than future vertices

```
def cycle_check(solver,v_from,v_to,name):
    solver.assert_and_track(v_from < v_to, name)
```

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 $a_{03} = 0$. 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:

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

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

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

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:

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

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.

4 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. In addition, we add a parameter s which represents the size of the range of possible weights on edges in a graph.

With these parameters we employ random generation to construct sample graphs. We are interested in the individual effect k, p and s 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.

Hi Mike. The rest of the section is describing each case individually and then research questions. We don't have these spelled out right now. What we hypothesize is that at some combination of k, p and s the solver runtime will become infeasible. Then we can make determinations on the viability of this method in different domains. For example, most tournaments will be a bounded number of participants, and score ranges, but may have maximum connectivity. Thus, we would be able to make a conclusion on the viability of our method for the tournament domain. As a stretch goal we wish to consider graphs of different distributions that Erdős-Rényi, e.g. Barabási-Albert or Watts and Strogatz.

5 RESULTS AND DISCUSSION

It was good enough to graduate

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

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

Module communicates with the backend solver to
    solve minimum feedback arc set problems

You'll notice that we thread the solver instance
    and cache through several
functions. This is purposeful to get as close as
    possible to referential
transparency in python. Think of it as a Reader
    Monad carrying the cache and
solver handle.

"""

from pprint import pprint
from copy   import deepcopy
import z3   as z

import gadgets as g
import graphs  as gs
import utils   as u


def find_cycle(cache,s,graph):
    """Find a cycle in a graph. s is a solver
        object, graph is an adjacency list
    e.g. {0:[0,1,2,3,4,5], 1:[2,3] ...}. Given the
         graph iterate over the
    graph, convert the key and values to symbolic
        terms and check for a cycle.
    Returns the unsat core as a list of strings e.
        g. [1->2, 2->3, 3->4, 4->1]
    """
    for source, sinks in graph.items():
        for sink in sinks:
            ## convert to strings
            s_source = str(source)
            s_sink   = str(sink)

            ## create symbolics
            sym_from = u.make_sym(cache,source)
            sym_to   = u.make_sym(cache,sink)

            ## lets go, this is just a fold over
                the dict yielding ()
            g.cycle_check(s, sym_from, sym_to, u.
                make_name(s_source,s_sink))

    r = []
    if s.check() == z.unsat: r = s.unsat_core()
    return u.parse_core(r)

def relax(graph, unsat_core, strategy):
    """Take a graph, an unsat-core and a strategy.
         use the strategy to find the
    edge to relax, relatx the edge and return the
        graph

    This function mutates graph
    """
    source, sink = strategy(unsat_core)
    outgoing = deepcopy(graph[source])
    print(graph[source])
    print(outgoing)
    print(sink in outgoing)
    print(outgoing.remove(sink))
    graph[source] = outgoing.remove(sink)
    if graph[source] is None: graph[source] = []

def go (cache,s,graph):
    # flag to end loop
    done = False

    # kick off
    while not done:
        pprint(graph)

        # get the core
        core = find_cycle(cache, s, graph)
        pprint(core)

        # if the core is empty then we are done,
            if not then relax and recur
        if core:
            # relax an edge by some strategy
            relax(graph,core,lambda x : x[0])
```

```
            # this is good enough for now but in
                the future we should try to
            # remove only the right constraint,
                which will be much much faster
            s.reset()
        else:
            done = True


def main():
    # just use a heterogeneous map as a cache
    cache = {}

    # spin up the solver
    s = z.Solver()

    go(cache,s,gs.round_robin_graph)
```

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

Module which defines sample graphs
"""


round_robin_graph = { "A": ["B","C","D","E","F"
    ],
                    "B": ["C","F"],
                    "C": ["D","E"],
                    "D": ["E","F"],
                    "E": ["B"],
                    "F": ["E","C"]
                  }

anti_greed_graph = { 0: [1],
                   1: [2,3],
                   2: [0],
                   3: [4],
                   4: [5],
                   5: [3,2]
}
```

```
"""
- 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 *

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

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

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 make_sym(cache, new, ty=Int):
    """Create a new symbolic variable in the
        backend solver. We use a cache to
    avoid repeated calls to the solver.
        Furthermore, because we are naming
    constraints we must ensure that we don't
        accidental use a duplicate name in
    the solver or else it will throw an exception
    """

    if new not in cache:
        sym_new = ty(str(new))
        cache[new] = sym_new

    return cache, cache[new]
```