A Symbolic Algorithm for Strongly Connected Component Analysis

Nagarajan Shanmuganathan

University of Colorado Boulder
`nagarajan.shanmuganathan@colorado.edu`

Abstract. The subject of this report is understanding why strongly connected decomposition is important and an efficient implementation of it for large graphs aids in the Büchi and Streett emptiness problems. An improvement over the classical Tarjan's algorithm for SCC decomposition has been implemented, known as *Lockstep*, which is a symbolic algorithm.

Keywords: Strongly connected components · Symbolic algorithms · Büchi and Streett Automata

1 Introduction

It has been shown that using Tarjan's classical algorithm for finding the strongly connected components of very large graphs is infeasible, even though its worst case run time in linear. This is because, Tarjan's algorithm is an *explicit* algorithm, which means that the algorithm operates on the explicit representation of the system. In this case, the algorithm has to consider and label each node individually.

In contrast, *implicit* or *symbolic* algorithms only use a set of predefined operations and do not explicitly access the system. The significance of symbolic algorithms in verification is as follows: to combat the state-space exploration, large systems must be succinctly represented implicitly and then symbolic algorithms are scalable, whereas explicit algorithms do not scale as it is computationally too expensive to even explicitly construct the system[1].

Although the worst-case time complexity of the symbolic algorithms is typically worse than that of corresponding explicit algorithms, they perform well as heuristics. This is one of the reasons that many large scale graph problems can be tackled only symbolically. Some of the important operations in symbolic graph algorithms are the set operations union, intersection, complementation and most importantly *image* and *preimage*. The latter two are collectively referred to as *steps*. The algorithm's efficiency in this project is defined by the number of steps, even though the number of steps is not the only factor that determines the time taken by a symbolic algorithm.

The remainder of this report covers some preliminaries, the main idea behind the algorithm, few results on the complexity of the algorithm and implementation details.

2 Preliminaries

This section discusses about some basic notations pertaining to graphs, symbolic algorithms, and Streett and Büchi automata.

2.1 Graphs and Strongly Connected Components

A (directed) *graph* is a pair $G = (V, E)$ where V is a finite set of nodes and $E \subseteq V \times V$ is a set of edges.

A *Strongly Connected Component (SCC)* of G is a maximal set $C \subseteq V$ such that for all $v, w \in C$ there is a path from v to w. An *SCC* C is *nontrivial* if for all $v, q \in C$ there is a nontrivial path from v to w. The SCC-quotient graph of (V, E) is a graph (V', E') with $V' = \{SCC(v) | v \in V\}$ and $E' = \{(C, C') | C \neq C'$ and $\exists v \in C, v' \in C' : (v, v') \in E\}$. A subset $S \subseteq C$ is *SCC-closed* if it is the union of set of SCCs.

2.2 Symbolic Algorithms

Symbolic algorithms operate on sets on nodes, represented by their characteristic functions. The notion of symbolic algorithms is formalized by the abstract data structure **Set**. A symbolic algorithm can manipulate sets using union, complementation, image and preimage. It can also test sets for inclusion, turn a node into a set, and pick an element, which returns an arbitrary element from the set. The important computations image and preimage are defined as follows:

$$img_G(S) = \{v' \in V | \exists v \in S : (v, v') \in E\},$$
$$preimg_G(S) = \{v' \in V | \exists v \in S : (v', v) \in E\}.$$

The time needed to compute an image or preimage is not constant. It depends on both the argument and the transition relation.

2.3 Streett and Büchi Automata

A *Streett automaton* is a tuple $\mathcal{A} = ((V, E), V_0, \mathcal{F})$ where (V, E) is a graph, $V_0 \subseteq V$ is the set of initial nodes, and $\mathcal{F} = \{(L_1, U_1), (L_2, U_2), \ldots, (L_k, U_k)\} \subseteq 2^V \times 2^V$ is the acceptance condition.

Strong Fairness Condition The Streett acceptance condition, also called *strong fairness*, requires that for every pair $(L, U) \in \mathcal{F}$, either L is visited only finitely often, or U is visited infinitely often. It allows one to state such properties as "if a request is infinitely often asserted, it is infinitely often granted".

The language of Street automaton $\mathcal{A}$ is nonempty if (V, E) contains a *fair cycle: a cycle* D such that for all pairs $(L, U) \in \mathcal{F}$, we have that $L \cap D \neq \emptyset$ implies $U \cap D \neq \emptyset$.

A *generalized Büchi automaton* is a Streett automaton $((V, E), V_0, \mathcal{F})$ such that for all pairs $(L, U) \in \mathcal{F}$ we have $L = V$.

Language and Fairness The language of a Büchi automaton is nonempty if there is a cycle that contains at least one node in U_i for every i. The emptiness condition for Büchi automata can be stated in terms of SCCs: the language is nonempty if the automaton contains an SCC C that is *fair*: it is nontrivial and for every i, $C \cap U_i \neq \emptyset$.

The nonemptiness check for Streett automata can also be based on the identification of the SCCs of the automaton graph. However, if an SCC C intersects L_i, but not U_i, it is still possible for it to contain a fair cycle, provided such a cycle does not contain any nodes in L_i.

3 Algorithm

The decomposition of a graph into SCCs in this project is dependent critically on *lockstep search*[2]. This section will deal with the idea behind the lockstep algorithm.

The idea is to pick a node v and then compute the set of nodes that have a path to v ($EF(v)$) and the set of nodes that have a path from v ($EP(v)$). The intersection of these two sets is the SCC containing v. Both the sets are SCC-closed and, after removal of the SCC, the algorithm proceeds recursively. This is then combined with lockstep search that performs backward and forward search simultaneously, until one converges. This gives an n log n bound.

The algorithm takes a Streett automaton and a set $P \subseteq V$ as arguments. The search is restricted to nodes in P, which is initially equal to V. If we set the acceptance condition to $\emptyset$, then all SCCs are fair, algorithm *Report* never recurs, and the algorithm performs a complete SCC decomposition. Fig. 1 shows the lockstep algorithm.

By randomly picking a node v (Line 12) from P, the algorithm computes the SCC(v). Sets F and B contain subsets $EP(v)$ and $EF(v)$, respectively. Until either $F = EP(v)$ or $B = EF(v)$, the algorithm successively approximates the sets (Lines 17-22). The intersection of B and F is a subset of the SCC containing v (Line 23).

The algorithm then refines the approximation of B, in which Ffront remains empty. The set Bfront contains the new nodes that reach v, and Bfront $\cup F$ contains the nodes of the SCC found in the current iteration. When this set is empty, the search terminates: all nodes in C are now in B, and the intersection of B and F is the SCC of v. $C = F \cap B$ is the computed, and then C is reported and then the algorithm recurs.

The recursion splits the graph into three parts: $F \backslash C, P \backslash F$, and C, and recurs on the first two sets.

If $P \neq \emptyset$ then it does not contain a fair SCC. If $P \neq \emptyset$, it contains a fair SCC if and only if that SCC is either C, or included in F or $P \backslash F$.

For Büchi emptiness, *Report* enumerates all fair SCCs, but it never recurs. When checking Streett emptiness, if *Report* finds a nontrivial SCC that does not imeediately yield a fair cycle, it removes the "bad nodes" and calls *Lockstep* on the rest of the SCC. Fig. 2 shows the *Report* algorithm.

```
algorithm Lockstep
in: Streett automaton $\mathcal{A} = ((V, E), V_0, \mathcal{F})$
    Set $P \subseteq V$

begin
    Node $v$;
    Set $F$, $B$;
    Set Ffront, Bfront, $C$, Converged;

    if $P = \emptyset$ then return;

    $v$ = pick($P$);

    $F := \{v\}$; Ffront := $\{v\}$;
    $B := \{v\}$; Bfront := $\{v\}$;

    while Ffront $\neq \emptyset$ and Bfront $\neq \emptyset$ do
      Ffront := (img(Ffront) $\cap P) \setminus F$;
      Bfront := (preimg(Bfront) $\cap P) \setminus B$;
      $F := F \cup$ Ffront;
      $B := B \cup$ Bfront
    od;

    if Ffront = $\emptyset$ then
      Converged := $F$
    else
      Converged := $B$;

    while Ffront $\cap B \neq \emptyset$ or Bfront $\cap F \neq \emptyset$ do
      Ffront := (img(Ffront) $\cap P) \setminus F$;    // One of these two fronts is empty
      Bfront := (preimg(Bfront) $\cap P) \setminus B$;
      $F := F \cup$ Ffront;
      $B := B \cup$ Bfront
    od;

    $C := F \cap B$;

    Report($\mathcal{A}, C$);

    Lockstep($((V, E), V_0, \mathcal{F})$, Converged $\setminus C$)
    Lockstep($((V, E), V_0, \mathcal{F}), P \setminus$ Converged)
end
```

Fig. 1. The Lockstep Algorithm

```
algorithm Report
in: Streett automaton $\mathcal{A} = ((V, E), V_0, \mathcal{F})$
    Set $C \subseteq V$,

begin
    Set $C'$;

    print SCC $C$ identified;

    if $C$ is trivial then return;

    if $\forall i : C \cap L_i \neq \emptyset$ implies $C \cap U_i \neq \emptyset$ then do
       print "SCC is fair; language is not empty";
       return
    od;

    $C' = C$;

    for each $i$ such that $C \cap L_i \neq \emptyset$ and $C \cap U_i = \emptyset$ do
       $C' := C' \setminus L_i$
    od

    if $C' \neq \emptyset$ then
       Lockstep$(((V, E), V_0, \mathcal{F}), C')$
end
```

Fig. 2. The Report Algorithm

4 Complexity of the Algorithm

This section will state the results of the *Lockstep* algorithm. Detailed proof of the theorem is given in [2].

Theorem 1. *Algorithm Lockstep runs in $\mathcal{O}(n(\log n + p))$ steps.*

Corollary 1. *If $p = 0$, Lockstep uses $\mathcal{O}(n \log(dN/n))$ steps.*

5 Implementation

This section will deal with the implementation details of the *Lockstep* and *Report* algorithms.

For the project, the algorithms were implemented and tested using Python. The implementation uses NetworkX [3] for dealing with most the graph operations. The functions getImage and getPreimage compute the images and preimages of the nodes that are present in Ffront. NetworkX's descendants and ancestors methods are used respectively to achieve that. They return the images and preimages in the form of python sets as shown in Listing 1.1.

```python
def getImage(G, Ffront):
  img_union = set()
  for node in Ffront:
    img = nx.descendants(G, node)
    img_union = img_union | img

  return img_union

def getPreimage(G, Bfront):
  preimg_union = set()
  for node in Bfront:
    preimg = nx.ancestors(G, node)
    preimg_union = preimg_union | preimg

  return preimg_union
```

Listing 1.1. Image and Preimage computation

The *lockStep* method calls these two functions. The *lockStep* method does exactly what the algorithm in Fig. 1 does. Line 6 picks the random vertex v and Line 44 computes the $C = F \cap B$. Line 45 makes a call to the *Report* method. Lines 48-49 make the recursive call after splitting the graph into three portions as described in Section 3.

```python
 1 def lockStep(G, nodes, initial_nodes, acceptance_condition,
       sccs):
 2   if len(nodes) == 0:
 3     return
 4 
 5   # Random sample of a vertex from the current set of nodes
 6   v = random.sample(nodes, 1)[0]
 7   print(len(nodes))
 8 
 9   F = { v }
10   Ffront = { v }
11   B = { v }
12   Bfront = { v }
13 
14   while len(Ffront) != 0 and len(Bfront) != 0:
15     img = getImage(G, Ffront)
16     img_intersection = img & nodes
17     Ffront = img_intersection - F
18 
19     preimg = getPreimage(G, Bfront)
20     preimg_intersection = preimg & nodes
21     Bfront = preimg_intersection - B
22 
23     F = F | Ffront
24     B = B | Bfront
25 
```

```python
  converged = set()
  if len(Ffront) == 0:
    converged = F
  else:
    converged = B

  while len(Ffront & B) != 0 or len(Bfront & F) != 0:
    img = getImage(Ffront)
    img_intersection = img & nodes
    Ffront = img_intersection - F

    preimg = getPreimage(G, Bfront)
    preimg_intersection = preimg & nodes
    Bfront = preimg_intersection - B

    F = F | Ffront
    B = B | Bfront

  C = F & B

  report(G, nodes, initial_nodes, acceptance_condition, C, sccs)

  lockStep(G, converged - C, initial_nodes, acceptance_condition, sccs)
  lockStep(G, nodes - converged, initial_nodes, acceptance_condition, sccs)
```

Listing 1.2. Lockstep function

Listing 1.3 shows the implementation of *report*. It prints out if the SCC is fair upon giving the acceptance conditions of the Streett automaton as one of the parameters.

```python
def report(G, nodes, initial_nodes, acceptance_condition, C, sccs):
  c_prime = set()

  print('SCC: ', C)

  if len(C) == 1 or len(C) == 0:
    return

  fair = True
  for condition in acceptance_condition:
    if len(C & { condition[0] }) != 0:
      if len(C & { condition[1] }) == 0:
        fair = False
        break

  if fair:
    print("SCC is fair; language is not empty")
    sccs.append(C)
    return

  c_prime = C

  for condition in acceptance_condition:
    if len(C & { condition[0] }) != 0 and len(C & { condition
   [1] }) == 0:
      c_prime = c_prime - condition[0]

  if len(c_prime) != 0:
    lockStep(G, c_prime, initial_nodes, acceptance_condition,
     sccs)
```

Listing 1.3. Report function

5.1 Results

Fig. 3 shows the result of running the program with a directed graph with 20 nodes and 41 edges.

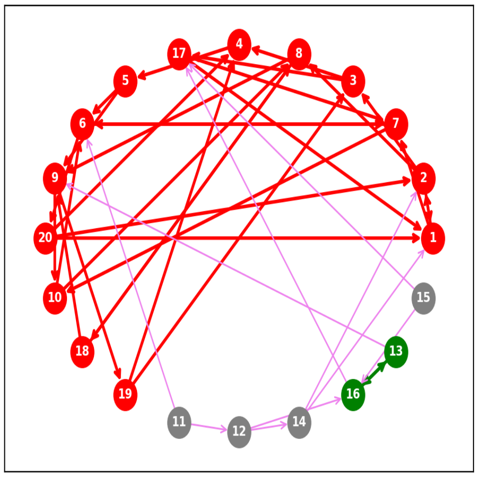

Fig. 3. SCC Decomposition

There are two nontrivial SCCs in this graph. The one that contains the vertices that are colored in green and the other SCC contains the vertices that are in red. The remaining vertices colored in gray are trivial SCCs.

Upon giving the Streett acceptance conditions as a separate input to the program, the fairness conditions and emptiness of the language will be returned by the *report* function.

The code and the input files can be found at [4].

6 Further Readings

In [2] there is a modification of the *Lockstep* algorithm that sharpens the bounds for Büchi emptiness. They use a trimming technique to achieve this, which computes $P \cap \text{img}(P) \cap \text{preimg}(P)$. They also present another variant of *Lockstep* called MaxSCC which finds the existence of a fair cycle.

Based on the SCC decomposition technique, [1] presents symbolic algorithms for graphs and Markov decision processes (MDPs) with strong fairness objectives. They present extensive comparisons between the explicit algorithms that are existing and the symbolic algorithms.

7 Conclusion

Several classes of ω-automata have been proposed in the literature. Especially Büchi automata, Rabin automata and Streett automata have been used extensively for succinct representation of languages. Faster and efficient algorithms to decompose of a graph into its Strongly Connected Components will aid in further research in analyzing these automata.

This project provided me with some insights on the automata and opened up a new problem space for me to explore - symbolic algorithms.

References

1. Krishnendu Chatterjee et al.,: Symbolic Algorithms for Graphs and Markov Decision Processes with Fairness Objectives. CoRR (2018)
2. Roderick Bloem, Harold N. Gabow, Fabio Somenzi: An Algorithm for Strongly Connected Component Analysis in $n \log n$ Symbolic Steps. Formal Methods in System Design (2006)
3. https://networkx.github.io/documentation/stable/
4. https://github.com/nagarajan-shanmuganathan/scc_decomposition