Minimizing LR(1) State Machines is NP-Hard

Wuu Yang

Computer Science Department
National Chiao-Tung University
Taiwan, R.O.C.

***Abstract*—LR(1) parsing was a focus of extensive research years ago. Though most fundamental mysteries have been resolved, a few remain hidden in the dark corners. The one we bumped into is the minimization of the LR(1) state machines, which we prove is NP-hard. It is the node-coloring problem that is reduced to the minimization puzzle. The reduction makes use of a technique in constructing a context-free grammar from the graph to be colored. The expected LR(1) state machine is derived from the constructed context-free grammar. A minimized LR(1) machine can be used to recover a minimum coloring of the original graph.**

Keywords—*graph coloring; LR(1) parser; LALR(1) parser; minimize LR(1) state machine; node coloring; NP-hardness; parsing.*

I. Introduction

Parsing is a basic step in every compiler and interpreter. LR parsers are powerful enough to handle almost all practical programming languages [11]. The downside of LR parsers is the huge table size. This caused the development of several variants, such as LALR parsers, which require significantly smaller tables at the expense of reduced capability.

The core of an LR(1) parser is a deterministic finite state machine. The LALR(1) state machine may be obtained by merging *every* pair of similar states(Note that two states are *similar* if and only if they become identical if the look-ahead sets in the items in the two states are ignored.) in the LR(1) machine [8]. In case (reduce-reduce) conflicts occur due to merging,(note that only reduce-reduce conflicts may occur due to merging similar states.) the parser is forced to revert to the larger, original LR(1) machine. Due to the significant size difference between LR(1) and LALR(1) state machines, we know there are many pairs of similar states in an LR(1) machine. If any pair of similar states may cause conflicts, the parser will be forced to use the much larger LR(1) machine. It would be more reasonable to merge some, but not all, pairs of similar states [16]. The result, called an *extended LALR(1) state machine*, is smaller than the LR(1) machine but larger than the LALR(1) machine.

For example, there are five pairs of similar states in the LR(1) machine in Figure 1. Only three pairs—$(s_1, t_1), (s_2, t_2), (s_3, t_3)$—can be merged. The pair of similar states—(s_5, t_5)—cannot be merged due to a (reduce-reduce) conflict. The last pair of similar states—(s_4, t_4)—cannot be merged because (s_5, t_5) are not merged for otherwise the resulting machine would become nondeterministic. Figure 2 is the corresponding (minimum) LR(1) machine.

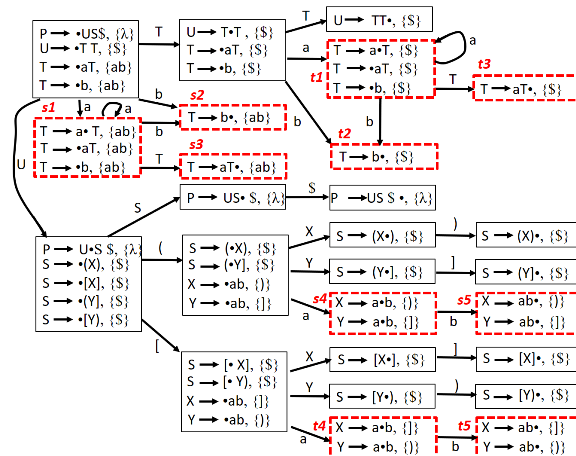

Fig. 1. The LR(1) machine of a grammar.

In general, two states in an LR(1) machine can be merged if and only if the following two conditions are satisfied:

1) The two states must be similar;
2) Corresponding successor states of the two states must have already been merged.

A further question is if there is an efficient algorithm that can merge the *most* number of similar states, thus producing a *minimum* LR(1) state machine. That is, we wish to minimize the LR(1) state machine. Since the number of similar states is finite, a naïve approach is to try all possibilities.

Our study shows that minimizing the LR(1) state machine is an NP-hard problem. We reduce the node-coloring problem to this minimization problem. Starting from an (undirected) graph to be node-colored, we construct a context-free grammar. Then the LR(1) machine of the context-free grammar is derived. We can use an algorithm to calculate the corresponding minimum LR(1) machine.

In order to recover a minimum coloring from this minimum LR(1) machine, we can perform one more easy step. In the LR(1) machine, every state that is not similar to any other states is removed, leaving only similar states. Then an edge between two similar states is added if the two similar states may cause conflicts. The resulting machine is called a *conflict graph*. Merging similar states in the LR(1) machine is essentially identical to merging states in the conflict graph. (Note that Due to the construction of the grammar, all states

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-634-7

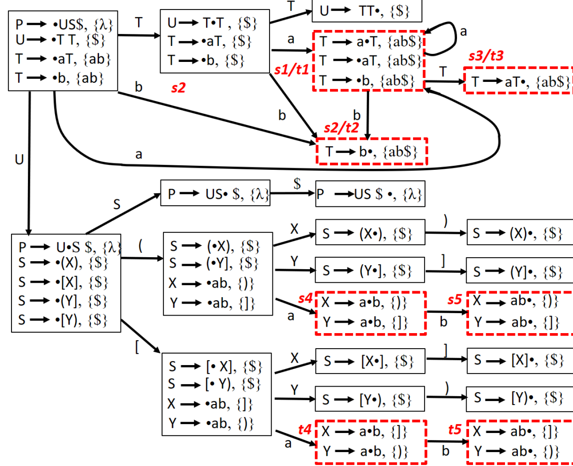

Fig. 2. The corresponding minimum LR(1) machine for Figure 1.

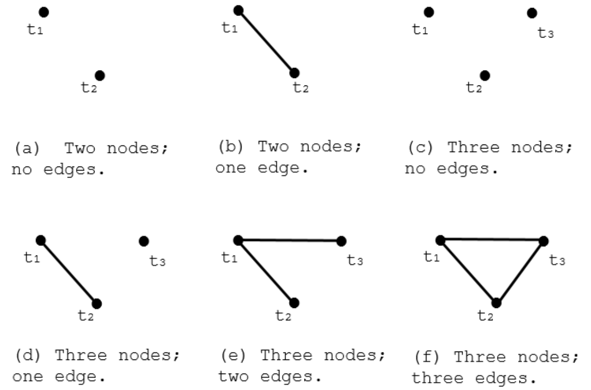

Fig. 3. Two cases for a color graph with two nodes and four cases for a color graph with three nodes.

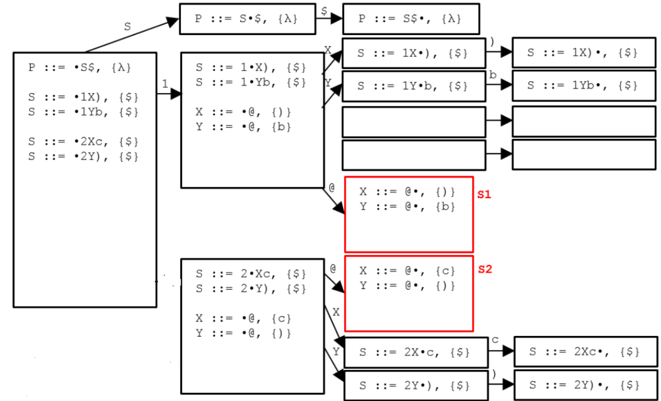

Fig. 4. The LR(1) machine for the graph in Figure 3 (b). Note that there is a single conflict $s_1 \leftrightarrow s_2$. The empty boxes are not part of this machine. They are used for comparison with later machines.

in the resulting conflict graph are similar to one another in the LR(1) machine. Furthermore, the conflict graph is actually isomorphic to the original color graph.) From the minimum LR(1) machine, it is straightforward to recover a minimum coloring.

The following theorem seems obvious but we wish to bring it to the reader's attention when reading this paper:

Theorem. *Let s_1, s_2, and s_3 be three similar states in an LR(1) machine. If the three states are not conflicting pairwise, then merging all three states will not create any conflicts.*

Due to the above theorem, we need to consider only pairs, not triples, quadruples, *etc.*, of similar states. This greatly simplifies our discussion.

Note also that there might be more than one minimum LR(1) machine for a given LR(1) machine.

LR parsers were first introduced by Knuth [11]. Since LR parsers are considered the most powerful and efficient practical parsers, much effort has been devoted to related research and implementation [1][3][7][10][12][14].

It is known that every language that admits an LR(k) parser also admits an LALR(1) parser [12]. In order to parse for an LR(1)-but-non-LALR(1) grammar, there used to be four approaches: (1) use the much larger LR(1) parser; (2) add ad hoc rules to the LALR(1) parser to resolve conflicts, similar to what yacc [10] does; (3) merge some, but not all, pairs of similar states [16]; and (4) transform the grammar into LALR(1) and then generate a parser. The transformation approach may exponentially increase the number of production rules [12] and the transformed grammar is usually difficult to understand. This paper shows that, although we wish to merge as many pairs of similar states as possible, this optimization problem is NP-hard.

Pager proposed two methods: "practical general method" (PGM) [14] and "lane-tracing method" [13] [15]. Chen [4] actually implements Pager's two methods as well as other improvements, such as unit-production elimination. Because the minimization problem is NP-hard, it is important to build practical LR parser generators. Pager and Chen's work is one of the best existing LR parser generators. The IELR method [5] includes additional capability to eliminate conflicts even if the grammar is not LR.

Both [14] and [5] attempt to find a *minimal* machine. However, *minimal* simply means "very small" or "locally minimum" rather than "globally minimum"[5]. This is different from our study of *minimization*.

The remainder of this paper is organized as follows. Section 2 will introduce the terminology and background. Section 3 introduces a reduction algorithm that translates an undirected graph into a context-free grammar and discusses the reduction of the coloring problem to the minimization problem. The last section concludes this paper.

II. Terminology and Background

A grammar $G = (N, T, P, S)$ consists of a non-empty set of nonterminals N, a non-empty set of terminals T, a non-empty set of production rules P and a special nonterminal S, which is called the start symbol. We assume that $N \cap T = \emptyset$.

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-634-7

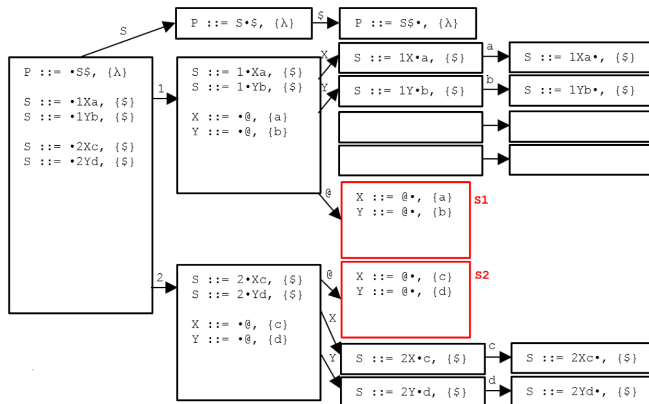

Fig. 5. The LR(1) machine for the graph in Figure 3 (a). There is no conflict in the machine. The empty boxes are not part of this machine. They are used for comparison with later machines. G32-0.

LR(1) parsing is based on a deterministic finite state machine, called the LR(1) machine. A state in the LR(1) machine is a non-empty set of items. An *item* has the form (A ::= $\gamma_1 \bullet \gamma_2$, la), where A ::= $\gamma_1\gamma_2$ is one of the production rules, $\bullet$ indicates a position in the string $\gamma_1\gamma_2$, and la (the *lookahead* set) is a set of terminals that could follow the nonterminal A in later derivation steps. The algorithm for constructing the LR(1) state machine for a grammar is explained in most compiler textbooks, for example, [2][8]. An example state machine is shown in Figure 1.

Two states in the LR(1) machine are *similar* if they have the same number of items and the corresponding items differ only in the lookahead sets. For example, states $s1$ and $t1$ in Figure 1, each of which contains three items, are similar states.

LR(1) state machines are closely related to LR(0) state machines. However, an LR(1) machine is much larger than the corresponding LR(0) machine because many similar states are introduced. In order to reduce the size of the LR(1) state machine, some or all pairs of similar states may be *merged* as long as no conflicts occur. For example, LALR(1) machines are obtained from LR(1) machines by merging *every* pair of similar states.

Sometimes merging two similar states may create a *(parsing) conflict*. The aim of *minimizing an LR(1) machine* is to merge as many pairs of similar states as possible without causing conflicts. Our study shows that this minimization problem is NP-hard.

III. REDUCTION

We may prove that minimizing LR(1) machines is an NP-hard problem by reducing the node-coloring problem to this minimization problem. Specifically, from a graph F to be colored, we construct a context-free grammar G. Then the LR(1) state machine M is derived from G. An algorithm is used to calculate the minimum state machine, from which a minimum coloring can be recovered.

(no edge)	(one edge)
P ::= S$	P ::= S$
S ::= 1Xa	S ::= 1X)
S ::= 1Yb	S ::= 1Yb
S ::= 2Xc	S ::= 2Xc
S ::= 2Yd	S ::= 2Y)
X ::= @	X ::= @
Y ::= @	Y ::= @

Fig. 6. The two grammars the color graphs in Figure 3 (a) and (b).

(a)	(b)	(c)	(d)
P ::= S$	P ::= S$	P ::= S$	P ::= S$
S ::= 1Xa	S ::= 1X)	S ::= 1X)	S ::= 1X)
S ::= 1Yb	S ::= 1Yb	S ::= 1Yb	S ::= 1Yb
S ::= 1Ze	S ::= 1Ze	S ::= 1Z=	S ::= 1Z=
S ::= 1Vf	S ::= 1Vf	S ::= 1Vf	S ::= 1Vf
S ::= 2Xc	S ::= 2Xc	S ::= 2Xc	S ::= 2Xc
S ::= 2Yd	S ::= 2Y)	S ::= 2Y)	S ::= 2Y)
S ::= 2Zg	S ::= 2Zg	S ::= 2Zg	S ::= 2Z=
S ::= 2Vh	S ::= 2Vh	S ::= 2Vh	S ::= 2Vh
S ::= 3Xi	S ::= 3Xi	S ::= 3Xi	S ::= 3Xi
S ::= 3Yj	S ::= 3Yj	S ::= 3Yj	S ::= 3Yj
S ::= 3Zk	S ::= 3Zk	S ::= 3Zk	S ::= 3Zk
S ::= 3Vm	S ::= 3Vm	S ::= 3V=	S ::= 3V=
X ::= @	X ::= @	X ::= @	X ::= @
Y ::= @	Y ::= @	Y ::= @	Y ::= @
Z ::= @	Z ::= @	Z ::= @	Z ::= @
V ::= @	V ::= @	V ::= @	V ::= @

Fig. 7. The four grammars constructed from graphs with 3 nodes by our algorithm.

In order to recover a minimum coloring, M can be simplified by removing every state that is not similar to any other state, resulting in a *conflict graph*. Merging similar states in the conflict graph is essentially identical to finding a minimum coloring of F.

We define a *node-coloring of a graph* as a partition of the set of nodes in the color graph satisfying the requirement that nodes connected by an edge cannot be in the same partition block. A *minimum coloring* is a partition with the fewest blocks. Similarly, a *merge scheme* of an LR(1) state machine is a partition of the states satisfying the requirement that states in the same partition block are similar to one another and do not conflict with one another. A *minimum merge scheme* is a partition with the fewest blocks. *Minimizing an LR(1) machine* is to find a minimum merge scheme of the machine.

We build a context-free grammar G for a given color graph F *inductively*.

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-634-7

Assume color graph F has n nodes. Then the constructed machine M has n states that are similar to one another; the remaining states are distinct and can be ignored in the discussion of merging similar states. There is 1-1 correspondence between the n nodes in F and the n similar states in M. We claim that M satisfies the following property:

> *F may be colored with k colors if and only if $n - k$ pairs of similar states in M may be merged (so that only k similar states remain).*

If there is any algorithm that can calculate the minimum LR(1) machine M_{min} from M by merging certain pairs of similar states, we can use that algorithm to solve the node-coloring problem—for all states that are merged into a single state in M_{min}, their corresponding nodes in F have the same color.

Due to the above property, we have successfully reduced the node-coloring problem to the minimization problem. Because the node-coloring problem is NP-hard [9], the minimization problem is also NP-hard.

To construct a context-free grammar from the color graph F, we first choose two arbitrary nodes t_1 and t_2. There are two cases, shown in Figure 3 (a) and (b): there is no or one edge t_1—t_2. Then one of the grammars in Figure 6 is selected.

Assume that there is an edge t_1—t_2 in F. Then the grammar on the right inFigure 6 is selected. The corresponding LR(1) machine is shown in Figure 4, in which there are two similar states (s_1 and s_2). Merging the two similar states will cause a conflict due to the terminal symbol ")". The grammar is carefully constructed so that the conflict $s_1 \leftrightarrow s_2$ corresponds to the edge t_1—t_2 in F.

In our constructed grammars, the numbers, such as 1, 2, 14, 27, *etc.*, are teminals and indicate a similar state in the resulting LR(1) machine and the order the corresponding nodes in F are chosen. The upper-case English letters, such as A, B, *etc.*, denote nonterminals. The lower-case English letters, such as a, b, *etc.*, denote terminals that are used only once in the grammar. These lower-case letters will not cause conflicts. The punctuation marks, such as ")" and "=", are terminals that will cause conflicts.

On the other hand, if t_1 and t_2 in F are not connected, the grammar on the left in Figure 6 will be selected. Figure 5 is the LR(1) machine for that grammar. There are two similar states in that machine (s_1 and s_2). The two similar states can be merged without conflicts. This corresponds to the fact that t_1 and t_2 in Figure 3 (a) can have the same color since there is no edge connecting them.

Note that the notion of *"two (similar) states can be merged"* in the LR(1) machine is closely related to the notion of *"two nodes can have there same color"* due to our construction.

The remaining nodes in F are chosen one by one in an arbitrary order. By adding one node at a time, we can gradually construct grammars $G_2, G_3, G_4, \ldots, G_n$. The complete algorithm for generating a context-free grammar from a graph is shown in Figure 10.

Example. The grammar on the right column in Figure 6 is extended to the grammar on the third column in Figure 7. The four grammars constructed from graphs with 3 nodes by our algorithm. (a) is for graphs with no edges (Figure 3 (c)); (b) is for graphs with one edge (Figure 3 (d)); (c) is for graphs with two edges (Figure 3 (e)); and (d) is for graphs with three edges (Figure 3 (f)). The production rules are classified into five categories. In particular, the boxed production rules in (c) are new rules added to the grammar on the right column in Figure 6.The corresponding color graphs are shown in Figure 3 (b) and (e), respectively. The boxed production rules are added by the algorithm in Figure 10. □

The grammar on the third column in Figure 7 is a typical grammar generated by our algorithm. The production rules are classified into five categories:

1) one starting production rule (*i.e.*, P ::= S \$)
2) four production rules of the form (Π ::= @)
3) four production rules whose right-hand sides begin with the terminal 1
4) four production rules whose right-hand sides begin with the terminal 2
5) four production rules whose right-hand sides begin with the terminal 3

The LR(1) machine is then derived from the grammar. We did not construct the LR(1) machines directly because context-free grammars are easier to generate.

Now consider the constructed LR(1) machine in Figure 9. Note that all items derived from rules of categories 1, 3, 4, and 5, appear only once in the whole LR(1) machine. Any state containing any of these items will not be similar to any other state and hence can be ignored. We could focus on states consisting solely of items derived from production rules of category 2. There are only 3 such states, which are indeed similar to one another. Each such state has all items of the form (Π ::= @ •, . . .), where Π is a nonterminal except P and S.

Another characteristic of the constructed LR(1) machine in Figure 9 is that there are no cycles. The longest path contains 3 steps.

In fact, all grammars generated by the algorithm in Figure 10 share the above characteristics. They help us to infer properties of the minimized LR(1) machines.

In the corresponding state machine in Figure 9, consider the four states that come immediately after the initial state. Items derived from rules in categories (3), (4) and (5) are cleanly separated because of the first symbols (which are integer terminals) on the right-hand sides of the rules. Hence, except the four states that come immediately after the initial state, those items whose first symbols on the right-hand sides are different will never be mixed in the same state in the state machine. Items derived from rules in category (2) are quite similar—actually all items of the form (Π ::= • @, where is Π is a nonterminal) appear in every state that comes immediately after the initial state (we will ignore the starting production

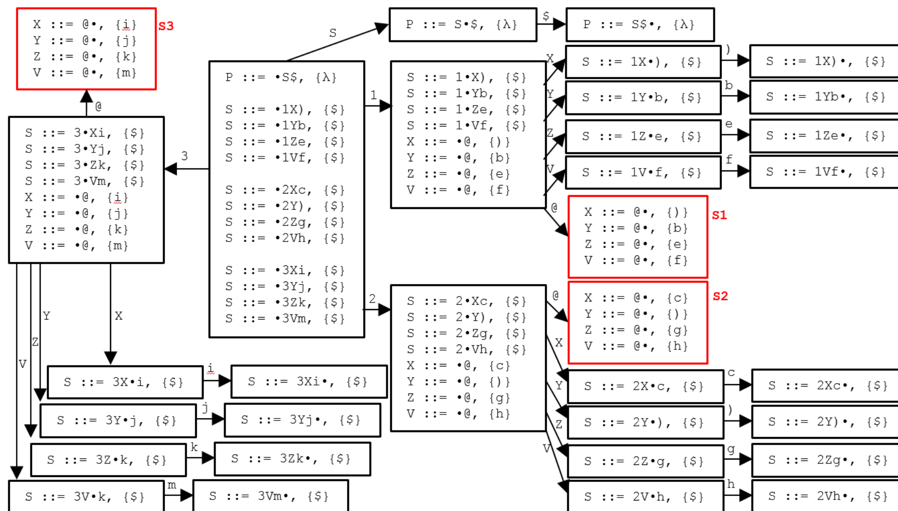

Fig. 8. The LR(1) machine for the graph in Figure 3 (d).

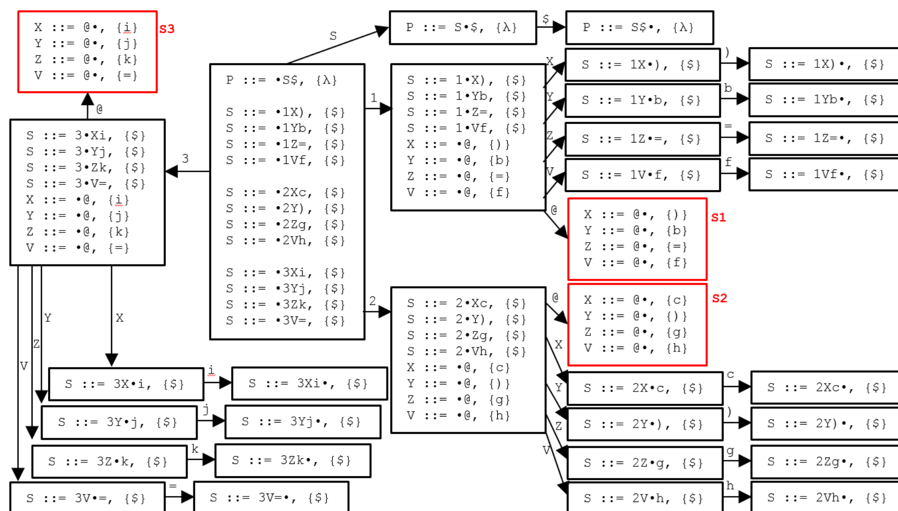

Fig. 9. The LR(1) machine for the graph in Figure 3 (e).

rule in this discussion). Furthermore, all items of the form (Π ::= @ •) appear in states that come two steps after the initial state. It is these states (which contain all items of the form (Π ::= @ •) and no other items) that are similar to one another. All other states are not similar to any other states and hence can be ignored when we discuss the merging of similar states.

Therefore, we can create or avoid conflicts among similar states by carefully adjusting the last terminal ψ in rules of the form (S ::= μ Π ψ), where μ is an integer terminal, Π is a nonterminal, and ψ is a terminal. In the four grammars in Figure 7, when ψ is a lower-case English letter (*e.g.*, "b" or "i"), that rule will not cause any conflict because the lower-case letter appears only once in the whole grammar. On the other hand, when ψ is a punctuation marks (*e.g.*, ")" or "="), a conflict is intentionally added to the grammar because that punctuation mark is used in two different production rules. The above discussion is related lines 18-22 in the algorithm in Figure 10.

We will use an algorithm to calculate the minimum state machine from M.

In what follows, we will describe how to recover a minimum coloring of the original color graph from the minimum state machine.

For the purpose of merging similar states, we may ignore all states that are not similar to any other states. To make conflicts among states explicit we add a *conflict edge* between two states if a conflict will occur when the two states are merged. The state machine in Figure 9 becomes Figure 11, which is called a *conflict graph*. In Figure 11, there are two conflict edges $s_1 \leftrightarrow s_2$ and $s_1 \leftrightarrow s_3$.

Finding a minimum merge scheme for the LR(1) machine is identical to finding a minimum merge scheme for the conflict graph. So we will focus on the conflict graph instead.

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-634-7

1. Nodes in graph P are listed as $v_1, v_2, \ldots, v_n$;
2. **if** there is an edge v_1—v_2 **then** G := the left grammar in Figure 6 ;
3. **else** G := the right grammar in Figure 6;
4. NewNonTerm := { X, Y };
5. **for** μ := 3 **to** n **do**
6. generate two new nonterminals, called Δ and Θ;
7. generate two new terminals, called ϕ and ω;
8. generate four production rules:
9. ("S ::= " μ Δ ϕ) and ("S ::= " μ Θ ω) and
10. (Δ " ::= @") and (Θ " ::= @");
11. **for** each nonterminal $\Pi \in$ NewNonTerm **do**
12. generate a new terminal, called ψ;
13. generate a production rule: ("S ::= " μ Π ψ);
14. **end**;
15. **for** δ := 1 **to** $\mu - 1$ **do**
16. generate two new terminals, called τ and ρ;
17. generate a production rule: ("S ::= " δ Θ ρ);
18. **if** there is an edge $v_\mu \leftrightarrow v_\delta$ **then**
19. /* *The following rule will cause a conflict due to ω.* */
20. generate a production rule: ("S ::= " δ Δ ω);
21. **else** /* *The following rule will NOT cause a conflict.* */
22. generate a production rule: ("S ::= " δ Δ τ);
23. **end**;
24. NewNonTerm := NewNonTerm $\cup$ $\{\Delta, \Theta\}$;
25. **end**;

Fig. 10. Algorithm for generating a context-free grammar from a graph. The four rules generated at lines 9, 17, and 20 will cause a conflict in the LR(1) machine due to the terminal ω.

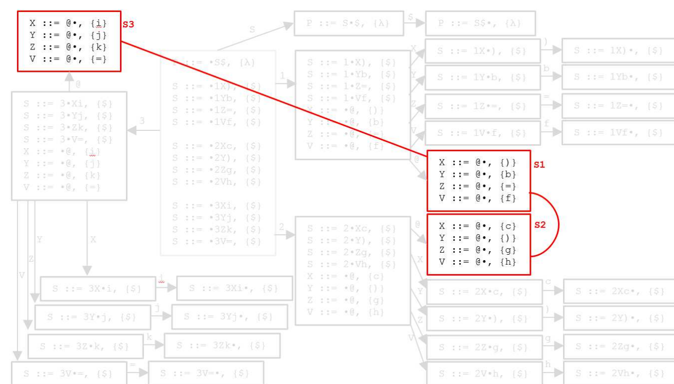

Fig. 11. The conflict graph. After removing the states that are not similar to any other states, only three states are left. We may add an edge $s_1 \leftrightarrow s_2$ to indicate there is a conflict edge $s_1 \leftrightarrow s_2$ and another conflict edge $s_1 \leftrightarrow s_3$.

The reader may find that the conflict graph (Figure 11) is isomorphic to the original color graph (Figure 3 (e)). This is due to the construction of the context-free grammar.

IV. Conclusion

We have reduced the node-coloring problem to the minimization problem of the LR(1) state machines. Therefore, the minimization problem is NP-hard.

There are efficient algorithms for minimization of finite state machines. LR(0) state machines are minimum by its construction. We show that LR(1) state machines cannot be easily minimized in general.

Note that minimizing an LR(1) machine is quite different from minimizing a general finite state machine. For one thing, we need to examine the items in the states of an LR(1) machine. On the other hand, minimizing a general finite state machine does not consider the "contents" of the states.

Acknowledgement

This work is supported, in part, by Ministry of Science and Technology, Taiwan, R.O.C., under contracts MOST 103-2221-E-009-105-MY3 and MOST 105-2221-E-009-078-MY3.

References

[1] A.V. Aho and S.C. Johnson. "LR Parsing," ACM Computing Surveys, vol. 6, no. 2, June 1974. pp. 99-124.
[2] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. Compilers: Principles, Techniques, and Tools. (2nd Edition) Prentice Hall, New York, 2006.
[3] T. Anderson, J. Eve, and J. Horning. "Efficient LR(1) parsers," Acta Informatica, vol. 2, 1973. pp. 2-39.
[4] X. Chen and D. Pager. "Full LR(1) Parser Generator Hyacc And Study On The Performance of LR(1) Algorithms." In Proc. 4th International C* Conf. Computer Science and Software Engineering (C3S2E '11), May 16-18, Montreal, CANADA, 2011. pp. 83-92.
[5] J.E. Denny and B.A. Malloy. "IELR(1) algorithm for generating minimal LR(1) parser tables for non-LR(1) grammars with conflict resolution," Science of Computer Programming, vol. 75, no. 11, November 2010. pp. 943-979. doi:10.1016/j.scico.2009.08.001.
[6] F.L. DeRemer. Practical translators for LR(k) languages. Project MAC Tech. Rep. TR-65, MIT, Cambridge, Mass., 1969.
[7] F.L. DeRemer. "Simple LR(k) Grammars." Comm. ACM, vol. 14, no. 7, July 1971. pp. 453-460.
[8] C.N. Fischer, R.K. Cytron, and R.J. LeBlanc, Jr.. Crafting A Compiler. Pearson, New York, 2010.
[9] M.R. Garey and D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, 1979. ISBN 0-7167-1045-5.
[10] S.C. Johnson. Yacc: Yet Another Compiler-Compiler. Bell Laboratories, Murray Hill, NJ, 1978.
[11] D. E. Knuth. "On the translation of languages from left to right." Information and Control, vol. 8, no. 6, July 1965. pp. 607-639. doi:10.1016/S0019-9958(65)90426-2.
[12] M.D. Mickunas, R.L. Lancaster, V.B. Schneider. "Transforming LR(k) Grammars to LR(1), SLR(1), and (1,1) Bounded Right-Context Grammars." Journal of the ACM, vol. 23, no. 3, July 1976. pp. 511-533. doi:10.1145/321958.321972.
[13] D. Pager. "The lane tracing algorithm for constructing LR(k) parsers." In Proc 5th Annual ACM Symp. Theory of computing, Austin, Texas, United States, 1973. pp. 172181.
[14] D. Pager. "A practical general method for constructing LR(k) parsers." Acta Informatica, vol. 7, no. 3, 1977. pp. 249-268. doi:10.1007/BF00290336.
[15] D. Pager. "The lane-tracing algorithm for constructing LR(k) parsers and ways of enhancing its efficiency." Information Sciences, vol. 12, 1977. pp. 1942.
[16] W. Yang, "Extended LALR(1) Parsing." In Proc. 13th International Conf. Autonomic and Autonomous Systems (ICAS 2017), May 21-25, 2017. Barcelona, Spain.