An Investigation into the Automatic Generation of Solutions to Problems in an Intelligent Tutoring System for Finite Automata

Nelishia Pillay
School of Computer Science
University of KwaZulu-Natal
Pietermaritzburg Campus
Pietermaritzburg, KwaZulu-Natal
South Africa
E-mail: pillayn32@ukzn.ac.za

Amashini Naidoo
School of Computer Science
University of KwaZulu-Natal
Howard College Campus
Durban, KwaZulu-Natal
South Africa
E-mail: Amashini.Naidoo@vodacom.co.za

Abstract

The study presented in this paper examines the possibility of the system itself generating solutions to problems presented to the learner in an intelligent tutoring system (ITS) for finite automata, rather than the lecturer having to create the solutions by hand and store them in a knowledgebase. The paper describes a genetic programming (GP) system for the evolution of deterministic finite automata (DFA). For each language the system takes a set of negative and positive sentences for that language as input and produces the corresponding DFA as output. The system was tested on 15 benchmark languages. The GP system was able to evolve solutions to all 15 languages. The paper also presents a comparison of the solutions generated by the system to "human" generated solutions.

Keywords: *intelligent tutoring systems, genetic programming, deterministic finite automata*

1. Introduction

Studies conducted (Cavalcante et al. 2004; Chesnevar et al. 2004; Rodger et al. 1998; Verma 2005; Viera et al. 2004) have revealed that students experience difficulties in learning formal language and automata theory (FLAT) concepts. According to Chesnevar et al. (2004) and Rodger et al. (1998) students find the abstraction of these concepts and their notation overwhelming. Furthermore, the traditional "pencil and paper" methods used to teach these topics do not provide the immediate feedback or visualization of concepts needed by learners and often lead to students becoming frustrated and disinterested in the course (Cavalcante et al. 2004; Verma 2005; Viera et al. 2004). This results in students formulating incorrect mental models of these concepts and hence developing a superficial understanding of the field.

A number of simulation tools, e.g. JFLAP, DEM, JCT, Language Emulator, have been developed to assist learners in overcoming these difficulties (Cavalcante et al. 2004; Verma 2005; Viera et al. 2004). While these simulators enable students to visualize the functioning of the automata and present them with the means of manually debugging their solutions, none of these tools provide the individualized tuition usually needed by learners to overcome such learning difficulties.

Intelligent tutoring systems (ITS) have been successfully employed in a number of subjects to provide such tuition by adapting their instruction of a particular domain to the needs of the learner (Freedman 2000). The study presented in this paper forms part of an effort aimed at developing an ITS for a third year undergraduate course on formal languages and automata theory. This paper investigates the possibility of the ITS inducing solutions to the problems it presents to the students, instead of these being created by the lecturer and stored in the knowledgebase of the ITS. The scope of the paper is restricted to deterministic finite automata (DFA). The paper evaluates genetic programming (GP) as a means of evolving DFAs. The main contribution made by the study presented in the paper is a genetic programming system that can evolve human-competitive DFAs. The next section provides an overview of GP and previous studies conducted to evolve DFAs.

The GP system implemented to induce these solutions is described in **Section 3**. **Section 4** reports on the performance of the system when applied to 15 benchmark problems and compares the evolved solutions to "human" generated solutions. Finally, **Section 5** summarizes the findings of the study and discusses future extensions of the project.

2. GP and Previous Work

Genetic programming is an evolutionary algorithm introduced by Koza that is based on Darwin's theory of evolution (Koza 1992). Essentially, the algorithm starts off with an initial population that is iteratively refined on successive generations until one of the elements of a generation is a solution or the best individual for a given problem. Each generation is created by applying one or more genetic operators such as crossover, mutation or reproduction. These operators are applied to parents selected using a selection method, e.g. tournament or fitness proportionate selection.

Although studies have been conducted to evaluate genetic algorithms as a means of evolving automata, the utilization of GP to induce finite state machines is still a relatively new area. Lankhorst (1995) and Dupont (1996) effectively make use of genetic algorithms in an attempt to solve the grammatical inference problem. In the first study Tomita's benchmark languages are used to test the genetic algorithm. Dupont also chose to use Tomita's benchmarks (1996), adding a few more languages to obtain a set of 15 languages. Brave (1996) attempted to evolve deterministic finite automata using a method that combines GP and cellular encoding. Cellular encoding is a GP technique that evolves the architecture, weights, and thresholds of a neural network, where evolution can take place concurrently. Brave tested his system on the seven Tomita languages and was successful in generating a solution to all seven languages except "L= |number of b's – number of a's| = 3n (multiple of 3)" for the alphabet • = {a, b}.

In order to prevent the pre-processing and post-processing needed to encode and decode the solution evolved by a genetic algorithm, it was decided to evaluate GP as a means of evolving the actual DFA. The system implemented for this purpose is described in the next section.

3. The GP System

Input to the GP system consists of the alphabet of the language and negative and positive sentences for the particular language and the output of the system is the corresponding DFA. The architecture of the system is described in this section.

3.1 Representation

A finite state machine is defined as a five-tuple (Q, • , •, q_0, F), where Q is the set of finite states, • is the set of input symbols (alphabet), • is the transition function, $q_0 \in$ Q is the start state, and F $\subseteq$ Q is the set of accepting states. The technique used here to induce finite state machines is different from the previous work done in this field in that firstly each individual of the population is represented as a directed graph. In addition to this the system implemented evolves the actual finite automaton and not instructions for the construction of the automaton. The system takes advantage of the natural graph based representation of a finite state machine. Each node of the graph is either an accept node or a reject node whereas each outgoing arc represents a symbol being read, i.e. an element of the language alphabet. Each individual generated is classed as a complete finite state machine if there is a transition from every state to some other state for every input symbol. The arity of each node is therefore the size of the alphabet. The first node chosen is the start state of the finite automaton. **Figure 1** shows an example representation of an individual.

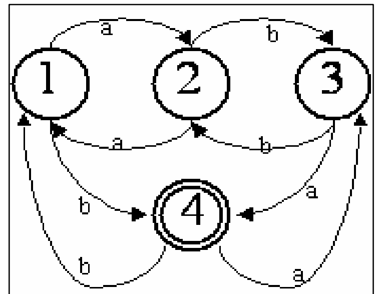


Figure 1: Example Representation of an Individual

3.2 Genetic Operators

The GP system uses reproduction, crossover and mutation to create the next generation. The genetic operators use tournament selection to choose the parents of the next generation. The reproduction operator makes an exact copy of the chosen parent and inserts this copy into the new population.

Crossover involves exchanging the genetic material of two randomly chosen parent individuals. The difference between crossover in this graph based system and a tree based system is that it is an exchange of subgraphs instead of subtrees. **Figure 2** below illustrates an example of how crossover is performed. In the example node 3 from parent 1 and node 2 from parent 2 are chosen as crossover points. The subgraphs are swapped with each other and the nodes are renumbered as shown below. If there are any outgoing arcs that are pointing at nodes that are no longer part of the graph, the arcs are randomly made to point at an existing node.

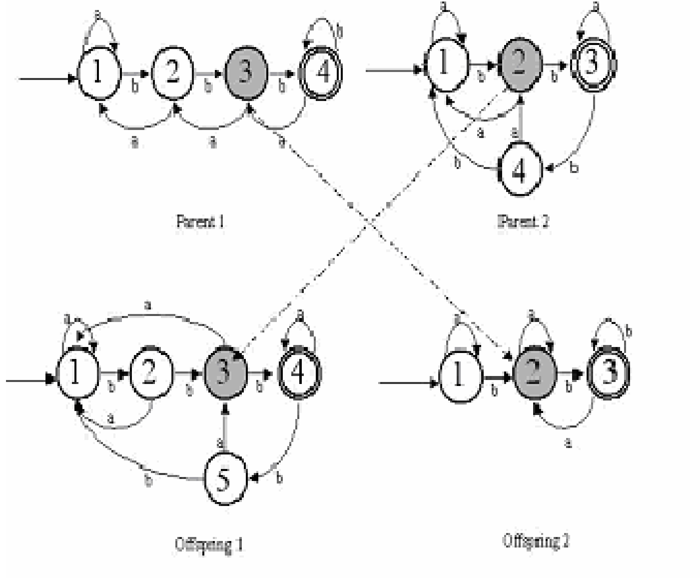


Figure 2: Crossover

Mutation is performed on only one individual. A node is randomly chosen in the individual and the subgraph rooted at that node is removed and replaced by a new randomly generated subgraph. As in crossover, if there are any outgoing arcs that are pointing to nodes that no longer exist after mutation occurs these arcs are made to point randomly to any existing nodes. **Figure 3** below shows an example of mutation.

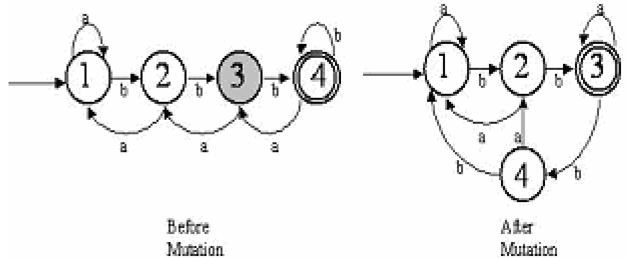


Figure 3: *Mutation*

The following section reports on the performance of this system when applied to the evolution of automata for 15 benchmark languages.

4. Results and Discussion

The genetic programming system described in **Section 3** was applied to the 15 languages in the language set listed in **Table 1**. The GP parameters used are listed in **Appendix A**. A sample of 200 sentences was used as input for languages L9 and L15 (these languages have a larger alphabet); 125 sentences for L10 and a 100 sentences for the rest of the languages. The system was able to generate solutions to all of the 15 benchmark languages. All DFAs were induced in less than a minute. These solutions (after being minimised in some cases) together with a corresponding "human generated" solution is listed in **Appendix B.** The JFLAP simulator was used to test the equivalence of these machines for each language. All 15 evolved DFAs were found to be equivalent to the corresponding "human generated" DFA. The DFAs evolved by the system for seven of the languages were exactly the same as the "human generated" solutions. The evolved DFAs for the remaining languages had more states than the corresponding "human generated" DFAs. The standard minimization algorithm was applied to these DFAs to remove the redundant states.

Table 1: *Language Set obtained from Coste (Coste 2006)*

Language	**Description**	**Example of a Positive Sentence**	**Example of a negative Sentence**
L1	a*	aaaaa	aaaaba
L2	(ba)*	bababa	ababa
L3	any sentence without an odd number of consecutive a's after an odd number of consecutive b's	abbabaa	abbabbbab
L4	any sentence over the alphabet a, b without more than two consecutive a's	abaabababb	ababaaaba
L5	any sentence with an even number of a's and an even number of b's	ababaabbbb	aabaaba
L6	any sentence such that the number of a's differs from the number of b's by 0 modulo 3	bbabbab	bbababba
L7	a*b*a*b*	aaabbbaabb	abbababa
L8	a*b	aaaaab	aaaaaba
L9	(a* + c*)b	aaaab, ccccb	abbbc, aaaba
L10	(aa)*(bbb)*	aaaabbb	aaaabb
L11	any sentence with an even number of a's and an odd number of b's	aaaba	aabba
L12	a(aa)*b	aaaaab	aaabaab
L13	any sentence over the alphabet a, b with an even number of a's	abbbabaa	bbabbbaba
L14	(aa)*ba*	aaaabaaa	abaaa
L15	bc*b + ac*a	bcccb, acca	acacc, abcac

5. Conclusions and Future Work

The main aim of the study presented in this paper was to test the feasibility of automatically generating DFAs to problems presented to a learner in an ITS for finite automata. These DFAs would be used in correcting the user's solution and generally providing feedback. Genetic programming was evaluated as means of evolving DFAs. The genetic programming system implemented was tested on 15 benchmark languages. The system was able to successfully evolve DFAs for all 15 languages. A comparison of the evolved solutions to "human generated" solutions revealed that in some cases the "machine generated" solutions contained more states. However, these can easily be removed by applying the standard minimization algorithm for removing redundant states from DFAs. Furthermore, the time taken to evolve each DFA was less than a minute. The only possible drawback of the system is the number sample sentences needed to generate each DFA. Investigations need to be conducted into whether the number of inputs can be reduced, how much effort is needed (is it more than constructing the DFAs by hand) to generate these sentences and whether this process can be automated.

6. References

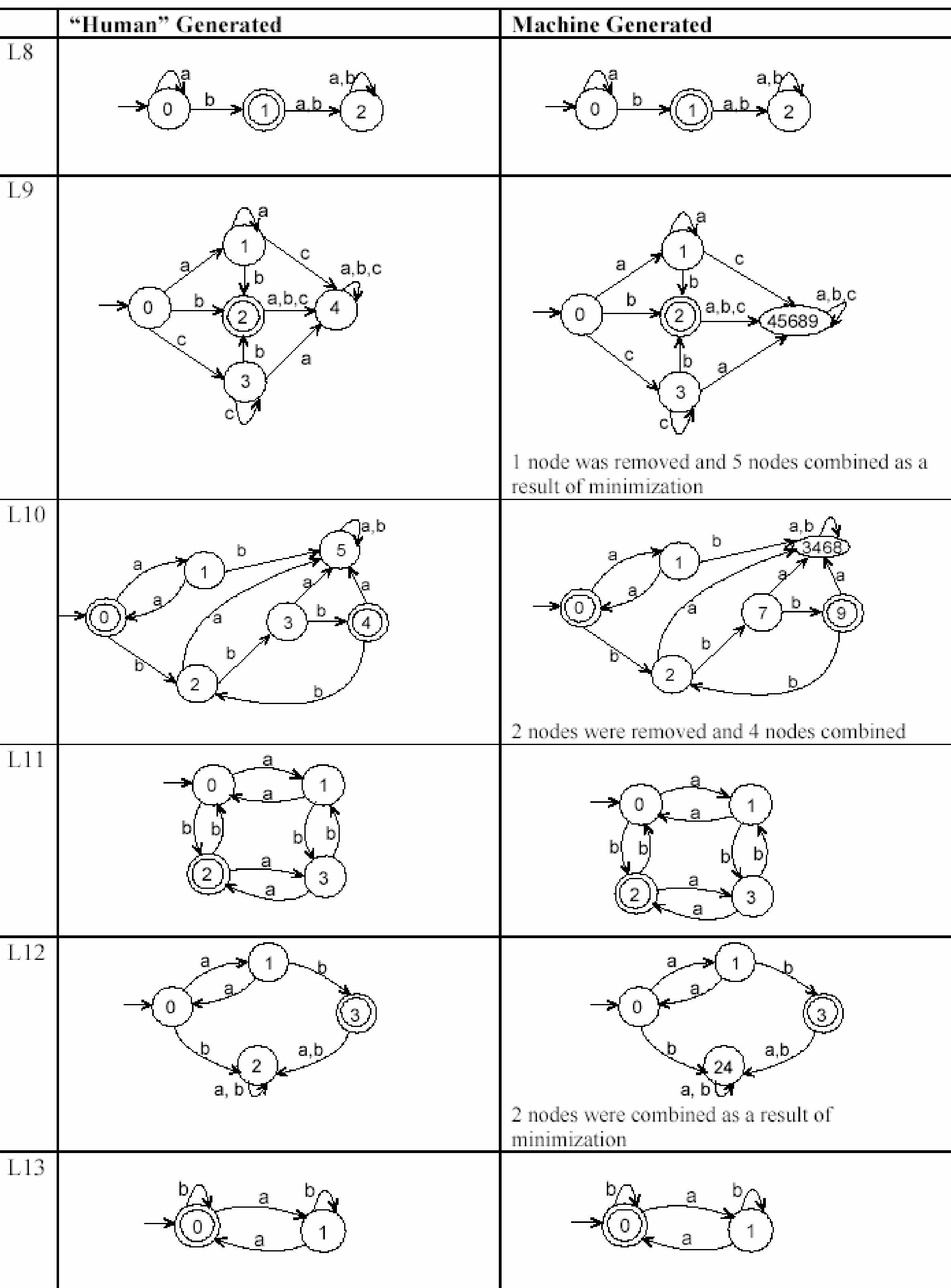
1. Brave, S. (1996). Evolving Deterministic Finite Automata Using Cellular Encoding. In *Genetic Programming 1996: Proceedings of the First Annual Conference* (eds. Koza, J.R., Goldberg, D. E., Fogel, D. B. and Riolo, R. L.), pp. 39 – 44, Standford University Press.
2. Cavalcante, R., Finley, T., Rodger, S. H. (2004). A Visual and Interactive Automata Theory Course with JFLAP 4.0. In *SIGCSE Bulletin inroads, Proceedings of SIGCSE '04*, Vol. 36, No. 1, pp. 140 – 144, ACM Press.
3. Chesnevar, C.I., Conzalez, M. P., Maguitman, A. G. (2004). Didatic Strategies for Promoting Significant Learning in Formal Languages and Automata Theory. In *SIGCSE Bulletin inroads, Proceedings of ITiCSE 2004*, Vol. 36, No. 3, pp. 7 – 11, ACM Press.
4. Coste, F. (2006). *Grammatical Inference Benchmarks Repository*, http://www.irisa.fr/symbiose/people/coste/gi_benchs.html.
5. Dupont, P. (1994). Regular Grammatical Inference from Positive and Negative Samples by Genetic Search: the GIG Method. In *Grammatical Inference and Applications, Second International Colloquium, ICGI-94*, 1994, 236 – 245, Berlin-Springer.
6. Freedman, R., Ali, S.S., McRoy, S. (2000). "What is an Intelligent Tutoring System". In *intelligence*, pp. 15 -16, Vol. 11, No. 3, ACM Press.
7. Koza J. R. (1992). *Genetic Programming I: On the Programming of Computers by Means of Natural Selection*, MIT Press.
8. Lankhorst, M. (1995). *A Genetic Algorithm for Induction of Nondeterministic Pushdown Automata.* Computer Science Report CS-R-9502, University of Groningen, Netherlands, http://citeseer.ist.psu.edu/lankhorst95genetic.html.
9. Rodger, S.H., Gramond. E. (1998). JFLAP: An Aid to Studying Theorems in Automata Theory. In *SIGCSE Bulletin inroads, Proceedings of ITiCSE '98*, Vol. 30, No. 3, pp. 302, ACM Press.
10. Verma R. M. (2005). A Visual and Interactive Automata Theory Course Emphasizing Breadth of Automata. *In SIGCSE Bulletin inroads, Proceedings of ITiCSE 2005*, Vol. 37, No. 3, pp. 325 – 329, ACM Press.
11. Vieira, L.F.M., Vieira, M. A. M., Vieira, N. J. (2004). Language Emulator, a Helpful Toolkit. *In the Learning Process of Computer Theory, in SIGCSE Bulletin inroads, Proceedings of SIGCSE'04* , Vol. 36, No. 1, pp. 135 – 139, ACM Press.

7. Appendix A: GP Parameters and Termination Criteria

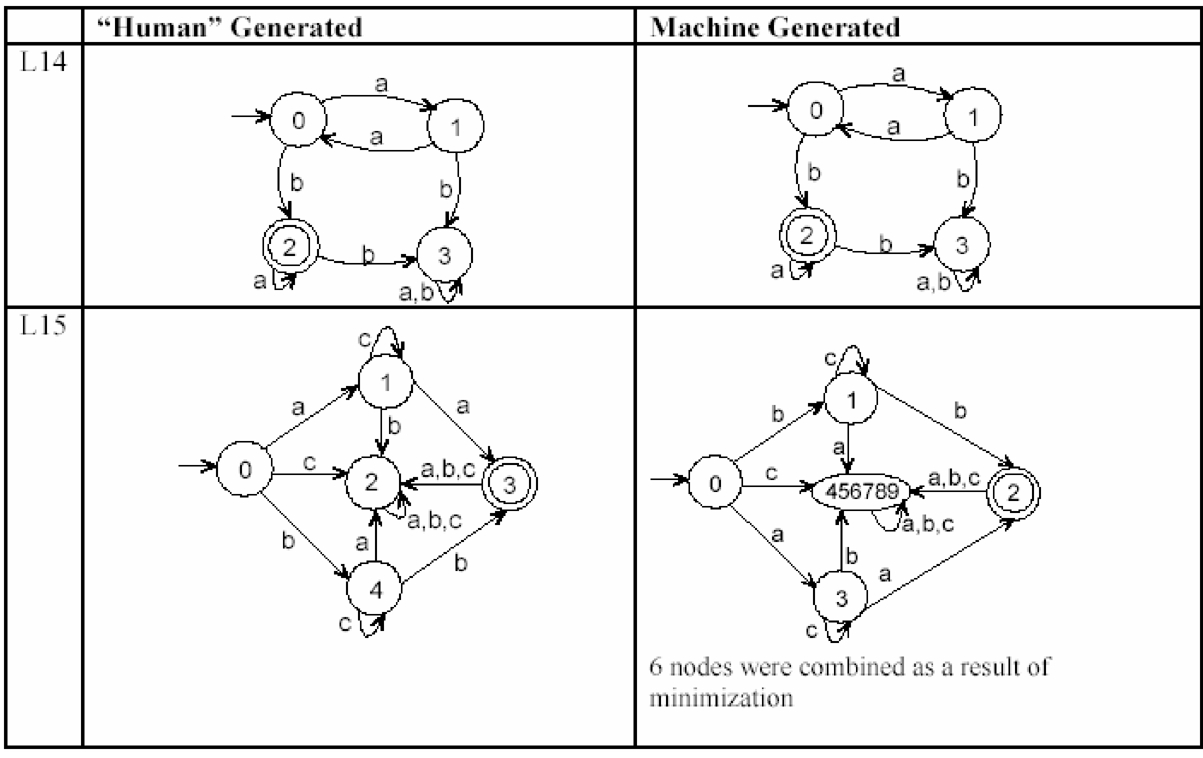
Objective	Evolve a finite state machine that accepts a certain language L
Population size	2000
Selection method	Tournament selection
Tournament size	5
Max number of Nodes	10
GP operator rates	Crossover=80%, Reproduction=10%, Mutation=5%
Maximum generations	50
Raw fitness	The number of correctly classified sentences.
Termination Criteria	A solution (i.e. correctly classifies all sentences) has been found or 50 generations are completed.

8. Appendix B: DFAs for the Fifteen Benchmark Languages

	"Human" Generated	Machine Generated
L1		
L2		
L3		5 nodes removed and 2 nodes combined as a result of minimization.
L4		2 nodes removed after minimization.
L5		
L6		
L7		2 nodes were removed and 3 nodes combined as a result of minimization



	"Human" Generated	Machine Generated
L8		
L9		1 node was removed and 5 nodes combined as a result of minimization
L10		2 nodes were removed and 4 nodes combined
L11		
L12		2 nodes were combined as a result of minimization
L13		



	"Human" Generated	Machine Generated
L14		
L15		6 nodes were combined as a result of minimization