Michael Curtis

Mike Kytka

Dan Mitchell

David Kloss

Forward Chaining

**The Algorithm:**

Forward Chaining is an algorithm that utilizes the repeated application of Modus Ponens to a set of Horn clauses to see whether or not a query can be known. A Horn clause is a set of at least one symbol conjunctions that imply exactly one other symbol. For example: A→ B, A ∧ B → C are both horn clauses, whereas A ∧ B → C ∧ D is not, since the implication is not a single symbol. The algorithm functions as follows:

Inputs:

clauses[] - a list of all propositional Horn clauses

known\_symbols[] - a set of symbols known to be true.

current\_symbol – a symbol that exists in known\_symbols that will be applied next.

Query – the symbol that forward chaining will check to see whether or not it can be definitively said that Query is true.

Return:

True if query can be definitively said to be true, false if known\_symbols runs out before query is added.

While known\_symbols is not empty, do{ //loops through the set of known symbols

current\_symbol = known\_symbols.pop(); //Pops a symbol from known\_symbols

if (current\_symbol == query){ //Checks if the current symbol equals the symbol we want to verify

return true;

}

for each clause in clauses{

if (current\_symbol is in clause){

clause.count - -;

if(clause.count == 0){

known\_symbols.push(clause.Implication)

clauses.delete(clause) //delete the current now empty clause

}

}

}

}

return false

**Proof of Correctness:**

Loop Invariant: The known\_symbols contains a list of symbols that are known to be true that have not been checked by the algorithm yet.

Initialization: The loop invariant is true initially as a precondition of the inputs. That is to say that a list of known symbols is supplied to the algorithm.

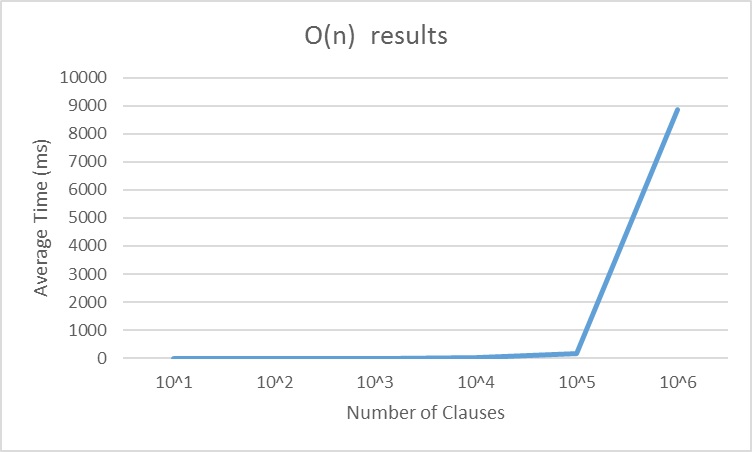
Maintenance: Upon entering the loop, we know the Loop Invariant to be true because it was true initially and nothing has changed. The algorithm then pops the next symbol to be analyzed off the list and checks to see if it equals the query. If they are equal, the algorithm returns true because the current\_symbol is guaranteed to be true by our Loop Invariant and therefore query is also true. If they are not equal, the algorithm decrements the count of every clause in which the current\_symbol appears by one. Decrementing the count represents the elimination of the known symbol from the premise which is valid for conjunctions. Therefore, when the count is zero, all premises have been known to be true and therefore Modus Ponens says the implication is true and therefore the implication is added to the list of known symbols. Otherwise, nothing is added to known\_symbols. Therefore, either no symbols have been added, or the symbols that have been added are known to be true. Both cases support the loop invariant.

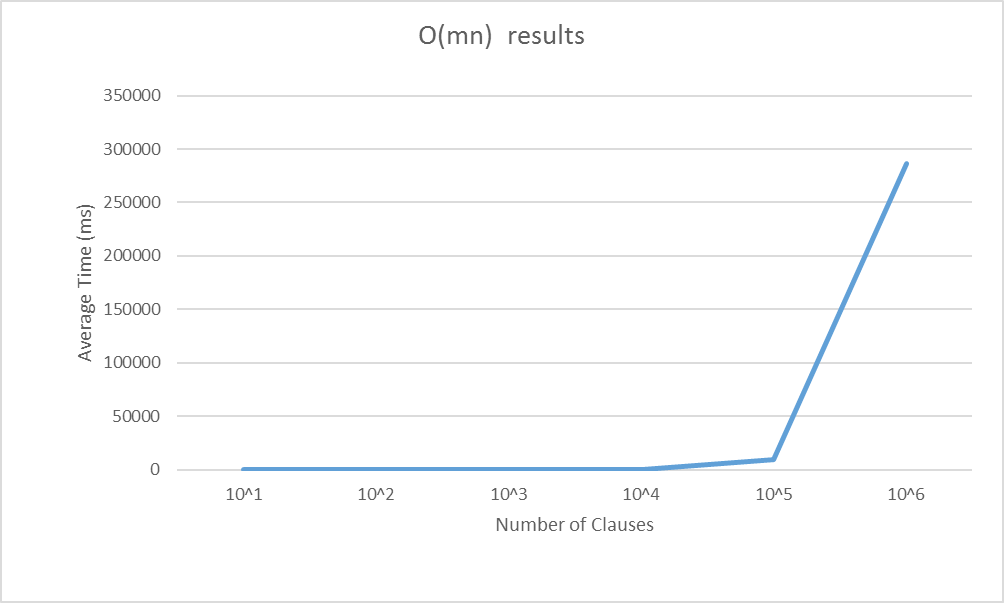
Termination: This algorithm is guaranteed to terminate because it is popping off a known symbol at every iteration, and every iteration will either add no new known symbols, or add n symbols to known symbols where n > 0, which means the algorithm must have deleted n clauses from the set of all clauses. In the case where no new symbols have been added, then known\_symbols is one closer to being empty which will terminate the loop. In the case where new symbols have been added and clauses have been deleted, then the algorithm is closer to the base case of analyzing one single clause, which is guaranteed to terminate. Since both move in discrete chunks with no possible way of recurring, the algorithm will terminate and the loop invariant holds.

**Analysis of Runtime:**

This algorithm has an average runtime of Θ (n \* m) where n is the number of clauses and m is the average length of each clause. This is because each clause needs to be analyzed (where n comes from) and during the analysis of each clause, each character must be checked for the current symbol (m). However, if a hash table is used rather than a list of character arrays, m becomes a constant which means the algorithm runs in Θ(n).

As we ran our five trials, we saw average numbers that represent what we expected from this algorithm. Using randomly generated files of length 10 through 10^6 of clauses that could be length 1 – 7, using 26 possible symbols.





Sure enough, both showed linear exponential sloping with an exponential scale, which indicates a linear runtime. The trial with O(mn) was slower, but could in theory process more clauses than the hash table because it has lower memory requirements.

**Conclusion:**

Forward Chaining is an algorithm that when hashing is used, the runtime is at worst O(n) where n is the number of symbols. However, this is under the assumption that accessing the hash is constant. In our case, using hashing greatly sped up our run time, but our hash function operated using symbols as keys, and a list of all their clauses as values. This means that there is still some iteration required to decrement the count of each row that a given symbol appears. It simply decreases the runtime by a factor of *s*, where *s* is the number of different symbols being used. For a randomly generated data-set, Forward Chaining can run in anywhere from 0 through n clauses before it returns true, and has to run through all known clauses before it can return false. This means returning true will always return at least as fast, but probably faster than returning false for a data set with the same number of clauses and input symbols.