Semester 1, 2022

Artificial Reasoning

Implementation of Inference Algorithms Based on Propositional Logic in a Digital Knowledge Base

Karan Manoj Shah (103141481)
Ruhan Venuja Nanayakkara (103522565)

Group ID: COS30019_A02_T033
Unit: COS30019 Introduction to Artificial Intelligence

Swinburne University of Technology

CONTENTS

INSTRUCTIONS

This inference engine is built for Microsoft Windows 10 operating system that has .NET Core 5.0 runtime libraries installed. It also shows full compatibility for Windows 11, Windows 8, and Windows 7.

The program is written in C# with the target framework of .NET 5.0 and compiled using Microsoft's Roslyn compiler.

A command-line interface (CLI) must be used to execute this program.

The program requires two arguments: **iengine <method> <filename>**
Example: **iengine TT test1.txt**
This instructs the program to read from file "test1.txt" and use the Truth-Table Model Checking algorithm.

The program offers the following inference methods:

- Truth Table Model Checking (TT): works for both generic proposition and Horn clause formats
- Forward Chaining (FC): works for knowledge bases in the Horn clause format
- Backward Chaining (BC) – works for knowledge bases in the Horn clause format
- WalkSAT (WSAT) – works for both generic proposition and Horn clause formats

The program has the following output if the arguments are valid:
YES: ***[no. of models of KB for TT*** *OR* ***list of facts inferred for F/BC*** *OR* ***no. of flips + true symbols for WSAT]***
OR **NO** (if query could not be entailed from KB)
For example:
YES: 3 (for TT) OR **YES: p1, p2, p3, q** (for FC/BC) OR **YES: 238 p1 p3 p2 p5 q d** (for WSAT)

Note: If using WalkSAT, you will be prompted to enter the maximum number of iterations when you initiate the program. The maximum number of iterations defines how many times the WalkSAT can randomly change the model and attempt inferring the query.

The data file containing propositional knowledge statements and query must have the following format:
TELL
[list of propositional sentences delimited by semicolon]
ASK
[a query]

If using Forward Chaining or Backward Chaining, the query must only be a single propositional symbol and the list of propositions can only be Horn clauses or facts. For Truth Table Model Checking and WalkSAT, the query and the list of propositions can be any valid propositional sentence.

Given below is a list of symbols that can be used to represent the logical connectives supported by this inference program:

Negation: **~**, Conjunction: **&**, Disjunction: **||**, Implication: **=>**, Biconditional: **<=>**.
The program also supports the use of parentheses **()** for construction of complex sentences.

For the Knowledge Bases supporting only Horn form, following are examples of valid sentences: **a** (a fact "a"), **a=>b**, **a & b => c**. The use of brackets is not supported for Horn form knowledge bases.

Propositional symbols must contain English alphabets and/or numbers only and can be of any length. Please note that the program is sensitive to the case of the propositional symbols.

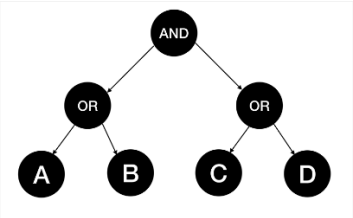
INTRODUCTION

Knowledge-based agents have proved to be valuable due to their innate ability to learn and grow with their environment. “Such agents combine and recombine information to suit myriad purposes […] Knowledge-based agents can accept new tasks in the form of explicitly described goals; they can achieve competence quicky by being told or learning new knowledge about the environment; and they can adopt to changes in the environment by updating the relevant knowledge” (Russell & Norvig, 2010).

The usefulness of knowledge-based agents justifies enabling the process of artificial reasoning and inference. While logic in general is a vast field, the scope of this project is on enabling machines to generate reasonings based on propositional logic. In this project we look at a range of ways to represent propositional knowledge and explore a multitude of inference algorithms that enable machines to derive knowledge from existing knowledge based on purely logical rules.

THE REPRESENTATION OF PROPOSITIONAL KNOWLEDGE

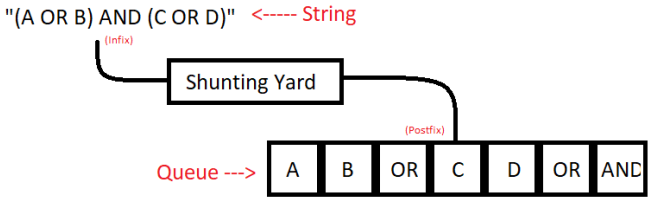
One of the toughest parts of this project was finding suitable data structures to store and manipulate propositional sentences. This is especially because we wanted to extend beyond just Horn clauses and allow computers to manipulate in general all kinds of propositional sentences: from just an atomic sentence or fact to the most complex statements.



Initially, we thought of using a **binary tree**, as this would **allow naturally the flow of hierarchy** in terms of logical connectives, and all the leaves could be directly comprehended as propositional symbols, so no expression checking would be required. Furthermore, the binary tree has **linear time complexity** when it comes to searching through, inserting, or deleting elements from it.

However, we realised that a binary tree **consumes more space than necessary** on the memory. This is because in addition to the payload, it also has to store two pointers. Furthermore, it also **requires a binary tree iterator object** which would consume more memory. This can become a problem when the program is to run on a lightweight node that works on high-volume knowledge sets.

Therefore, we chose to use a **queue structure**. This means the worst-case **look up time is linear** (same as binary tree), but the **insertion time is constant** (beating binary tree). Furthermore, it would require **significantly less memory than binary tree** when using large knowledge sets.



Then, the only issue remains is how to solve the logical connectives hierarchy issue? Unlike a binary tree, a queue **does not have natural support for hierarchies**; therefore, the only way to achieve hierarchies is to use an external support structure. We decided to use the **Shunting Yard algorithm** for this problem. Not only does this algorithm allow us to convert any kind of propositional sentence into a **postfix sentence** which directly solves the hierarchy challenge, but it also then makes it **easier to resolve the sentences when making inferences**, and we would simply deque elements one-at-a-time and assign them their values or operations on-the-fly, which ended up being a far more lucrative solution than implementing a binary tree and a binary tree iterator in terms of most space and time complexity.

IMPLEMENTATION

Provided below are the pseudocodes for the inference methods used in this assignment to highlight their implementation in the program.

function TT-ENTAILS?(KB, α) **returns** *true* or *false*
  inputs: KB, the knowledge base, a sentence in propositional logic
    α, the query, a sentence in propositional logic

  $symbols \leftarrow$ a list of the proposition symbols in KB and α
  return TT-CHECK-ALL($KB, \alpha, symbols, \{\}$)

function TT-CHECK-ALL($KB, \alpha, symbols, model$) **returns** *true* or *false*
  if EMPTY?($symbols$) **then**
    if PL-TRUE?($KB, model$) **then return** PL-TRUE?($\alpha, model$)
    else return *true* // when KB is false, always return true
  else
    $P \leftarrow$ FIRST($symbols$)
    $rest \leftarrow$ REST($symbols$)
    return (TT-CHECK-ALL($KB, \alpha, rest, model \cup \{P = true\}$)
      and
      TT-CHECK-ALL($KB, \alpha, rest, model \cup \{P = false\}$))

Figure 1: Truth Table Model Checking Algorithm Pseudocode (Russell & Norvig, 2010)

function PL-FC-ENTAILS?(KB, q) **returns** *true* or *false*
  inputs: KB, the knowledge base, a set of propositional definite clauses
    q, the query, a proposition symbol
  $count \leftarrow$ a table, where $count[c]$ is initially the number of symbols in clause c's premise
  $inferred \leftarrow$ a table, where $inferred[s]$ is initially *false* for all symbols
  $queue \leftarrow$ a queue of symbols, initially symbols known to be true in KB

  while $queue$ is not empty **do**
    $p \leftarrow$ POP($queue$)
    if $p = q$ **then return** *true*
    if $inferred[p] = false$ **then**
      $inferred[p] \leftarrow true$
      for each clause c in KB where p is in c.PREMISE **do**
        decrement $count[c]$
        if $count[c] = 0$ **then** add c.CONCLUSION to $queue$
  return *false*

Figure 2: Forward Chaining Algorithm Pseudocode (Russell & Norvig, 2010)

Function FOL-BC-Ask(*KB,goals,@*) **returns** a set of substitutions
inputs: *KB*, a knowledge base
goals, a list of conjuncts forming a query (@ already applied)
@, the current substitution, initially the empty subs. { }
local variables: *answers*, a set of substitutions, initially empty { }

if *goal*s is empty **then return** {@}
q' ← Subst(@,First(goals))
for each sentence r **in** *KB* where STANDARDIZE-APART(*r*)=(p_1 ^…^p_n →*q*) and @' ← Unify (*q,q'*) succeeds
new_goals ← [p_1, lll, p_n|Rest(*goals*)]
answers ← FOL-BC-Ask(*KB, new_goals*, Compose(@',@)) U *answers*
return *answers*

Figure 3: Backward Chaining Algorithm Pseudocode (Amarant, 2004)

Note that in our program, the backward chaining algorithm was slightly modified to avoid the infinite recursion problem by checking a list of already explored symbols and avoiding looping through those again.

function WALKSAT(*clauses*, *p*, *max_flips*) **returns** a satisfying model or *failure*
inputs: *clauses*, a set of clauses in propositional logic
p, the probability of choosing to do a "random walk" move, typically around 0.5
max_flips, number of value flips allowed before giving up

model ← a random assignment of *true*/*false* to the symbols in *clauses*
for each $i = 1$ **to** *max_flips* **do**
if *model* satisfies *clauses* **then return** *model*
clause ← a randomly selected clause from *clauses* that is false in *model*
if RANDOM$(0, 1) \leq p$ **then**
flip the value in *model* of a randomly selected symbol from *clause*
else flip whichever symbol in *clause* maximizes the number of satisfied clauses
return *failure*

Figure 4: WalkSAT Algorithm Pseudocode (Russell & Norvig, 2010)

EXTENSIVE TEST CASES

Here, we put to display an extensive range of test cases that test not just the edge scenarios but those particular datasets which highlight interesting information about the inference algorithms implemented.

Note that for WalkSAT, we are just mentioning output in terms of "YES" and "NO", because WalkSAT randomly chooses models which may or may not return an output, therefore there is no way to determine a one, true output for WalkSAT algorithm.

Horn-form Test Cases

Test Case 1:

Tell: a & b => p3; p2 & a => p3; c & p1 => p2; p3 & a => p1; c & a => p3; a; c;
Ask: p2

	Truth Table	Forward Chaining	Backward Chaining	WalkSAT
Intended Output	YES: 2	YES: a, c, p3, p1, p2	YES: c, a, p3, p1, p2	YES
Actual Output	YES: 2	YES: a, c, p3, p1, p2	YES: c, a, p3, p1, p2	YES

This specific test case was used as it includes recursion at multiple levels. This should cause Backward Chaining to loop over values that have already been explored. However, since we had added conditions to check if the function is rechecking previously explored values, our program will ignore previously explored clauses. By doing this, the function will proceed to check if the conclusion is satisfiable by another clause.

Test Case 2:

Tell: p1=> p3; p3 => d; d => p1; p2 & d => p3; p3 & p1 => p2; d;
Ask: p2

	Truth Table	Forward Chaining	Backward Chaining	WalkSAT
Intended Output	YES: 1	YES: d, p1, p3, p2	YES: d, p1, p3, p2	YES
Actual Output	YES: 1	YES: d, p1, p3, p2	YES: d, p1, p3, p2	YES

This test case includes recursion at the first level. Once again, the Backward Chaining algorithm will go into a recursive loop, constantly checking for values that have been explored. In our program, the conditions that have been added to check for recursive clauses will catch this and decided to ignore the clause as it has been explored previously.

Test Case 3:

Tell: p1 => c; c & d=> p2; p3 => p1; p2 => p3; c; d;
Ask: p1

	Truth Table	Forward Chaining	Backward Chaining	WalkSAT
Intended Output	YES: 1	YES: c, d, p2, p3, p1	YES: c, d, p2, p3, p1	YES

Actual Output	YES: 1	YES: c, d, p2, p3, p1	YES: c, d, p2, p3, p1	YES

This is an example of a false test case that the inference engine will have to deal with regularly. Our program successfully produces the expected outcomes for all the algorithms.

Test Case 4:

Tell: c => a; a => b; p2 => a; p2 => c; p1 => p3; p1; c;
Ask: p2

	Truth Table	Forward Chaining	Backward Chaining	WalkSAT
Intended Output	NO	NO	NO	YES
Actual Output	NO	NO	NO	YES

This is an example of a false test case that the inference engine will have to deal with regularly. Our program successfully produces the expected outcomes for all the algorithms. Note that WalkSAT generates a "YES" because it does not check for cases where the KB is satisfied but the query is false (which indicates no entailment), the WalkSAT algorithm will just check for a single model where it can find KB and query to both be satisfied and return that, therefore, the answer should be YES.

Test Case 5:

Tell: of => May; First => of; the => First; Born => On; Jason => Was; On => the; Was => Born; Jason;
Ask: May

	Truth Table	Forward Chaining	Backward Chaining	WalkSAT
Intended Output	YES: 1	YES: Jason Was Born On the First of May	YES: Jason Was Born On the First of May	YES
Actual Output	YES: 1	YES: Jason Was Born On the First of May	YES: Jason Was Born On the First of May	YES

This test case was used as the clauses in this test case form a complete sentence. Therefore, the output produced by our program will also be a fully formed sentence if the query is satisfied. Note that the permutations in both FC and BC will be identical for this test case.

Test Case 6:

Tell: of => May; First => of; the => First; Born => On; Jason => Was; On => the; Was => Born; the => and; Not => Second; Jason;
Ask: Second

	Truth Table	Forward Chaining	Backward Chaining	WalkSAT
Intended Output	NO	NO	NO	YES
Actual Output	NO	NO	NO	YES

This is another false test case that can be easily identified by reading the set of clauses. The inference engine produces a false outcome for this test case as it is the desired outcome.

Test Case 7:

Tell: a => b; a => c; c & b => a; c;
Ask: a

	Truth Table	Forward Chaining	Backward Chaining	WalkSAT
Intended Output	NO	NO	NO	YES
Actual Output	NO	NO	NO	YES

This is another false test case. It is false as a cannot be inferred from the clauses given c.

Test Case 8:

Tell: a => b; b => c; c => d; d => e; e => a; e => f; f => b; f => g; f => h; h;
Ask: g

	Truth Table	Forward Chaining	Backward Chaining	WalkSAT
Intended Output	NO	NO	NO	YES
Actual Output	NO	NO	NO	YES

This is a false test case which also includes recursion. Our program successfully avoids looping through the clauses infinitely and find that the query is not satisfied.

Test Case 9:

Tell: a => b; a => c; a => d; a => e; b => a; c => a; d => a; e;
Ask: a

	Truth Table	Forward Chaining	Backward Chaining	WalkSAT
Intended Output	NO	NO	NO	YES
Actual Output	NO	NO	NO	YES

This test case, too, is recursive for the Backward chaining algorithm. Once again, our program avoids running into such recursive loops and find that the query is not satisfied.

Generic Test Cases

Test Case 1:

Note that this test case is not used against WalkSAT as WalkSAT requires at least 1 clause in TELL.
Tell:
Ask: a || ~a

	Truth Table Model Checking
Intended Output	YES: 2
Actual Output	YES: 2

As shown by this test case, an interesting finding about Truth Table method is that it is the only inference algorithm explored in this assignment which does not require any clauses in the knowledge base to start making inferences. Since truth table model checking combines the list of symbols from the KB and the query, even if a symbol exists in the query which is not found in the KB, Truth Table can make an inference.

If the KB has no clauses, then the function PLTrue will always return true for KB for any given model. This means that if the query is true for all models where KB is true (and KB is always true, so the query must be a tautology), then the truth table method returns a YES.

Test Case 2:

Note that this test case is not used against WalkSAT as WalkSAT requires at least 1 clause in TELL.
Tell:
Ask: a

	Truth Table Model Checking
Intended Output	NO
Actual Output	NO

In this particular instance, the answer should be NO because the symbol "a" is not a tautology, whereas in test case 1 "a || ~a" was a tautology. Since "a" is not always true, the query is not always true with the KB is satisfied, therefore, the KB does not entail the query and the result is "NO".

Test Case 3:

Tell: (b & (~a => c) <=> c) || (d & (a <=> e || b)); c; d;
Ask: e

	Truth Table	WalkSAT
Intended Output	NO	YES
Actual Output	NO	YES

The answer here for Truth Table is interesting. Note that if we change the query from "e" to "~e" (not e), the output will still be "NO". This means that in every true model of KB, "e" does not have a fixed value, it can be either true or false depending on the values of the other variables, hence, KB does not entail the query.

Test Case 4:

Tell: a => b => ~a; b;
Ask: a

	Truth Table	WalkSAT
Intended Output	NO	NO
Actual Output	NO	NO

This is one of the few test cases where even WalkSAT cannot produce an affirmative response. This is because in this case, the KB and the query are mutually exclusive, so no model exists such that KB and query are both satisfied.

Test Case 5:

Tell: (~a => b) & (~a => c) & (~a => d)
Ask: a

	Truth Table	WalkSAT
Intended Output	NO	YES
Actual Output	NO	YES

FEATURES & BUGS

Features

All features specified in the assignment sheet are fulfilled in the assignment. No specification is missing.

Good programming practices have been followed when writing the code. Each function, code block, and ambiguous or logic-intensive code has been commented to provide clarity and make it easier to go through the program source files. Camel casing has been used to name files, functions, and classes as is the convention in C#.

The program provides the following inference algorithms (with their method codes):

- Truth Table Model Checking (TT)
- Forward Chaining (FC)
- Backward Chaining (BC)
- WalkSAT (WSAT)

The Knowledge Base is initialised by reading data from an external file which is specified through a CLI argument. The file both TELLs propositional sentences which are added to the KB and ASKs a propositional symbol or sentence which serves as the query for the inference engine.

For Forward Chaining & Backward Chaining, the inference engine takes in Horn clauses only and the query may only be a propositional symbol. For Truth Table Model Checking and WalkSAT, the sentences TELLed and ASKed can be generic propositional sentences or Horn-form clauses.

Generic inference algorithm (TT and WSAT) supports all the logical operatives, as specified below:

- *Negation* ~
- *Conjunction* &
- *Disjunction* ||
- *Implication* =>
- *Bidirectional* <=>
- *Parentheses* ()

Algorithms only supporting Horn-form clauses only parse implication and conjunction.

If inference is successful, Forward Chaining and Backward Chaining show as output a list of propositional symbols that they have inferred (in order) while inferring the query. Truth Table Model Checking shows the number of models of KB, whereas WalkSAT shows the number of flips it performed to find a suitable model and then enumerates the symbols inferred. If inference fails, all algorithms simply print out “NO”.

Another feature in this project is extensive testing has been performed considering not just edge cases of each algorithm, but also various scenarios that allowed us to discover interesting features about the search algorithms, as discussed in detail in the *Extensive Test Cases* section.

Bugs

In some test cases, the KB does not entail the query because while there are some models where the KB and query are both true, there are other models where the KB is true, but the query is false, which means the KB does not entail the query.

However, since WalkSAT works on randomly setting variables in the propositional sentences, sometimes it finds a model where KB and query are both true and returns that model, even though there may exist such a model where KB is true and query is false, which means the correct output should be "NO".

This isn't really a bug in that this is how random SAT solvers work, they aim to find some scenario where both the query and the KB are satisfied. However, this doesn't necessarily mean that the KB entails the query, which sometimes leads to false outputs for test cases where this is the case.

Another bug is that while we have implemented the WalkSAT algorithm such that it supports both Horn clauses and generic clauses, there is a component of the WalkSAT algorithm where it has to determine how important the symbols in a clause are to the KB as a whole. This particular part of the WalkSAT algorithm works correctly only for the Horn clause input format, whereas for generic knowledge bases it will return values which may not always be correct.

This is because – originally – WalkSAT is meant for CNF format only (and since CNF has a natural relation to Horn Clause, we can extend WalkSAT to work perfectly for Horn clauses, as we have done in this project). This means that WalkSAT must not be used for generic clauses, however, we realised that we can very easily enable WalkSAT to parse and manipulate generic clauses and added that as an extension to our work. Note that this bug does not mean WalkSAT does not find solutions in generic test cases, it just means that it can take a longer time when working with generic clauses than Horn clauses.

RESEARCH & EXTENSIONS

The inference engine program was enhanced by making the following research extensions:

- Added support for generic propositional sentences
- Implemented the WalkSAT algorithm
- Modified WalkSAT algorithm to support generic sentences
- Modified Backward Chaining algorithm to avoid infinite recursions
- Carried out extensive tests to provide interesting data about the algorithms

The extensive testing which allowed for identification of interesting data regarding the algorithms can be found in the *Extensive Test Cases* section.

The pseudocode for implementation of the WalkSAT algorithm can be found in the *Implementation* section. Interesting features about the WalkSAT algorithm has also been listed in the *Extensive Test Cases* section. Note that WalkSAT is an algorithm designed only for CNF clauses. Since CNF clauses and Horn clauses have a natural relation, this means WalkSAT can be extended to work perfectly for Horn clauses, as we have achieved in this project. However, we extend beyond just implementing WalkSAT, we also enabled it to parse and work with generic clauses. Please note that there is still a component of the WalkSAT algorithm which does not fully support generic clauses. This does not mean that the WalkSAT will not find the solution, however, it certainly can take a much longer time when working with generic sentences due to this issue. The specifics regarding this bug can be found in the *Features & Bugs* section.

The WalkSAT implementation on CLI is slightly different than the other inference algorithms. You can start WalkSAT method by using the following CLI Argument: **iengine WSAT <filename>**. When you do this, you will then be asked to "**Specify Maximum Iterations**", which should be a positive integer that tells WSAT the maximum number of times it can randomly flip the model and check for query entailment. If within that limit the algorithm is unable to find a match, it returns "**NO**". If a match is found while within the

limits, the output is of the following nature: "**YES: [number of iterations] [list of true symbols in model]**". Here, the "number of iterations" is how many iterations did it take for the algorithm to come to the solution. The "list of true symbols" is just all the symbols of the model that have evaluated value of true.

Backward Chaining works by finding a conclusion that matches the query, then exploring each of the symbols in the premise of that clause. Sometimes, this causes an infinite recursion as a conclusion occurs in the premise of one of the symbols in its own premise. In this assignment, we enhanced the backward chaining algorithm to work with even those test cases that may lead to recursion. This was achieved by simply maintaining a list of explored nodes. This list was updated when a new node was inferred. When the BC algorithm made a recursive call, we pass this list on as an argument, so that each recursive instance knows to avoid those clauses which will throw the algorithm in a never-ending recursive loop. The recursive test cases and corresponding test results can be found in the *Extensive Test Cases* section.

TEAM SUMMARY REPORT

Component	Karan	Ruhan	Component Weightage
Knowledge Base	50%	50%	10%
Shunting Yard & Postfix Operations	50%	50%	10%
Truth Table Model Checking	60%	40%	25%
Forward Chaining	60%	40%	20%
Backward Chaining	60%	40%	20%
WalkSAT	40%	60%	15%
Overall Contribution to Assignment:	55%	45%	100%

Table 1: Team Member Contribution Breakdown

The team members were always updated with the other teammate's work thanks to a communication channel opened on Discord. The work was also regularly pushed to GitHub to keep all changes and updates transparent and to keep record of who worked on what and maintain a timeline for the code.

Both team members are satisfied with the work of the other and agree that they were treated with respect throughout the course of the assignment. Both teammates took feedback seriously received by the other and actively worked upon the same.

ACKNOWLEDGEMENTS

- **Charles Harold:** For offering clarity on the myriad data structures that can be used to represent propositional logic and brilliant guidance throughout the development of the assignment.
- **Qinyuan Li:** For providing detailed information on various inference algorithms, their benefits and limitations, and for providing much needed support throughout the course of the project.
- **Bao Vo:** For explaining in detail various concepts key to this assignment including propositional logic and myriad inference algorithms, and providing inspiration, support, and outstanding lecture material which proved crucial to this assignment.

REFERENCES

Amarant, T., 2004. *Backward Chaining,* Providence: University of Rhode Island.

Russell, S. & Norvig, P., 2010. *Artificial Intelligence: A Modern Approach.* 3rd ed. New Jersey: Pearson Education Inc.