
UNIVERSITY OF CAPE TOWN


DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Project Final Paper 2022

Title: Scalable Defeasible Reasoning V2: Scalability of Lexicographic Closure

Author: Dhiresh Thakor Vallabh

Project Abbreviation: SCADR2

Supervisor(s): Professor Thomas Meyer

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	0
Theoretical Analysis	*0*	*25*	15
Experiment Design and Execution	*0*	*20*	10
System Development and Implementation	*0*	*20*	10
Results, Findings and Conclusions	*10*	*20*	10
Aim Formulation and Background Work	*10*	*15*	15
Quality of Paper Writing and Presentation	*10*		10
Quality of Deliverables	*10*		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	*0*	*10*	
Total marks		**80**	

Scalable Defeasible Reasoning V2: Scalability of Lexicographic Closure

Project Paper

Dhiresh Thakor Vallabh

University of Cape Town

Cape Town, South Africa

THKDHI001@myuct.ac.za

ABSTRACT

Knowledge representation and reasoning is a field of artificial intelligence (AI) that formally represents and enables reasoning of information about the world. Information is stored in a knowledge base, from which conclusions are drawn based on the information provided. Current reasoning systems based on classical propositional logic are able to perform working implementations of reasoning correctly and efficiently. However, as more information about the world is added to these knowledge bases, instances of contradictory information is more likely to occur. Defeasible reasoning is a form of non-monotonic reasoning that allows the reasoner to retract previously held beliefs when given contradictory information. The knowledge base is able to accept, and handle, information that is "typically" true, but not necessarily always true. This allows AI systems to mimic human behaviour through atypical reasoning scenarios. Lexicographic closure is a method of defeasible entailment proposed by Kraus, Lehmann and Magidor. Defeasible entailment describes what should defeasibly follow from a knowledge base containing classical and defeasible information. Some progress has been made to create systems that incorporate defeasible reasoning. However, there are still many improvements to be made to ensure the scalability of these systems. This paper builds on previous work in this regard to obtain more efficient algorithms for defeasible reasoning. This focuses on the design and integration of algorithms for the lexicographic closure approach. This investigation will evaluate and compare the theoretical time and space complexities and the practical execution times of these algorithms.

CCS CONCEPTS

• **Theory of computation → Automated reasoning**; • **Computing methodologies → Non-monotonic, default reasoning and belief revision**.

KEYWORDS

artificial intelligence, knowledge representation and reasoning, defeasible reasoning, Boolean satisfiability solving

1 INTRODUCTION

Knowledge representation and reasoning (KRR) is a fundamental part of artificial intelligence. KRR utilizes symbols to represent knowledge in which inferences are made and new elements of knowledge are formed about the world. To represent reasoning, we will be differentiating between propositional reasoning and defeasible reasoning. Propositional reasoning is monotonic as previously drawn conclusions must not be contradicted by new information [16]. These contradictions can allow for all information to be derived, making the knowledge base useless. To illustrate this logic, we will use the common example, "Penguins can fly." We, as humans, know that penguins cannot fly. However, a knowledge base may only know that birds have wings. Penguins are birds as they have wings, therefore penguins can fly. Propositional logic is not fully capable of mimicking human reasoning and is limited by the lack of exceptions in the knowledge base. Thus, this research focuses on non-monotonic reasoning, that allows for elements of knowledge to be considered typically true [3]. Defeasible reasoning addresses atypical reasoning scenarios in which humans fundamentally think differently, which may be due to their beliefs, context, or other circumstances. The reasoner can temporarily dismiss commonly held beliefs when presented with new contradictory information.

The work in Sections 1, 2, and 6 were completed jointly with Evashna Pillay.

2 BACKGROUND

2.1 Propositional Logic

Propositional logic [16] is a framework that represents information about the world as logical statements known as *formulas*. A formula consists of *propositional atoms* which can be assigned *truth values* (true or false). Formulas use *connectives* to connect propositional atoms ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$). With $\alpha, \beta \in \mathcal{L}$ where $\mathcal{L}$ is the set of all formulas we can create $\neg\alpha$, $\alpha \wedge \beta$, $\alpha \vee \beta$, $\alpha \rightarrow \beta$, $\alpha \leftrightarrow \beta$. An *interpretation* is a total function $I : P \mapsto \{T, F\}$ that takes a formula, $\alpha \in \mathcal{L}$, and assigns each propositional atom in α a truth value. The set of all of these interpretations is represented as $\mathcal{W}$. The truth value of an interpretation, created by α, is denoted as $I(\alpha)$. If $I(\alpha)$ is true for the formula α, then I satisfies α. This is denoted as $I \Vdash \alpha$. A *knowledge base* is a finite set of propositional formulas. A knowledge base, $\mathcal{K}$, is satisfied by an interpretation, I, if and only if for every $\alpha \in \mathcal{K}$, I satisfies α. If at least one interpretation satisfies $\mathcal{K}$, then that I is a *model* of $\mathcal{K}$.

2.2 Entailment

A knowledge base, $\mathcal{K}$, *entails* a formula, α, (written as $\mathcal{K} \models \alpha$) when every model of $\mathcal{K}$ is satisfied by α. Propositional entailment is monotonic. This means that all logical statements and conclusions

in the knowledge base are infallible. This means that any new contradictory information that is introduced can never be made as an exception [18]. However, the knowledge base must still be able to handle this new information and could potentially derive any formula. This renders the knowledge base generation useless which is why a nonmonotonic approach is required [12] [14]. The KLM style framework is our preferred choice.

When checking if $\mathcal{K} \models \alpha$, we can use a satisfiability (SAT) solver [18]. The SAT solver checks the satisfiability of $\mathcal{K}$ by checking if the models of $\mathcal{K}$ satisfy $\mathcal{K} \cup \{\neg\alpha\}$. This is discussed further in the Relevant Works section.

2.3 KLM Approach to Defeasible Reasoning

The KLM Approach models its statements in the form of $\alpha \mid\sim \beta$. This is interpreted as "α typically implies β". This means that we can conclude β from α unless contradictory information is received. There are multiple methods used to deduce information from a defeasible knowledge base, this notion is known as defeasible entailment. Defeasible entailment determines whether a knowledge base entails a defeasible implication. This approach adheres to the rationality properties defined by Lehmann and Magidor [11]. Two methods which fall within the KLM approach are rational closure and lexicographic closure. This paper will discuss both methods but will focus on the lexicographic method.

2.4 Rational Closure

Rational closure is an LM-rational method [8] of defeasible entailment proposed by Lehmann and Magidor [8]. Before providing an overview on rational closure, the following concepts will be defined; minimal ranked entailment, materialisation and Base Rank Algorithm.

2.4.1 Minimal Ranked Entailment. Ranked entailment is similar in principle to ranked interpretations, the lower the ranked model the more likely there exists a minimum ranked model in the ordering. This principle involves a partial order for all ranked models of knowledge base K and ranks the interpretations by how likely it is. The rational closure of a knowledge base K is based on determining the minimum ranked model [8], this must satisfy some defeasible implication for the knowledge base K to entail it [4].

2.4.2 Materialisation. The material counterpart of a defeasible implication $\alpha \mid\sim \beta$ is the propositional formula $\alpha \rightarrow \beta$ [5]. The material counterpart of a defeasible knowledge base $\mathcal{K}$ is denoted as $\overrightarrow{\mathcal{K}}$ and represents the set of material counterparts, $\alpha \rightarrow \beta$, for every defeasible implication $\alpha \mid\sim \beta \in \mathcal{K}$.

2.4.3 Base Rank Algorithm. The initial step performed on a knowledge base $\mathcal{K}$ is the base rank algorithm [17]. This determines the minimum ranked model.

Pseudo-code of the algorithm is shown below:

(1) Divide the statements into the classical statements $\mathcal{K}_C$ and the defeasible statements $\mathcal{K}_\mathcal{D}$.

(2) Materialise $\mathcal{K}_\mathcal{D}$ into $\overrightarrow{\mathcal{K}}_\mathcal{D}$. $\overrightarrow{\mathcal{K}}_\mathcal{D}$ in $\mathcal{E}_0^K$ and $\mathcal{K}_C$ as is.

(3) For each $\alpha \rightarrow \beta \in \mathcal{E}_0^K$, determine if $\mathcal{E}_0^K \cup \mathcal{K}_C \models \neg\alpha$.

(4) If true, α is exceptional and all formulas within $\mathcal{E}_0^K$ with α are moved to $\mathcal{E}_1^K$.

(5) Rank $\mathcal{R}_0^K$ is assigned to the formulas $\mathcal{E}_0^K \backslash \mathcal{E}_1^K$.

(6) Repeat the process for the next subset, i.e., $\mathcal{E}_1^K$ until no more formulas need to be assigned.

(7) Add the final rank $\mathcal{R}_\infty^\mathcal{K}$ to store $\mathcal{K}_C$.

Algorithm 1: Base Rank

Data: A knowledge base $\mathcal{K}$
Result: An ordered tuple $(R_0, R_1, \ldots, R_{n-1}, R_\infty, n)$

```
i := 0;
E_0 := →K;
while E_{i-1} ≠ E_i do
    E_{i+1} := {α → β ∈ E_i | E_i ⊨ ¬α};
    R_i := E_i \ E_{i+1};
    i := i + 1;
end
R_∞ := E_{i-1};
if E_{i-1} = ∅ then
    n := i − 1;
else
    n := i;
end
return (R_0, R_1, ..., R_{n-1}, R_∞, n);
```

2.4.4 Rational Closure Algorithm. $\mathcal{R}^\mathcal{K}$ from the base rank algorithm will be used, given a knowledge base $\mathcal{K}$ entails some defeasible implication $\alpha \mid\sim \beta$.

Pseudo-code of the algorithm is shown below:

(1) Check $\overrightarrow{\mathcal{K}}$ entails $\neg\alpha$.

(2) If false, α is satisfied with $\mathcal{K}$. Check that $\overrightarrow{\mathcal{K}}$ entails $\alpha \rightarrow \beta$.

(3) If true, α is unsatisfied with $\mathcal{K}$. The most preferred rank is withdrawn from $\mathcal{R}^\mathcal{K}$. The resulting knowledge base will be denoted as $\mathcal{K}'$.

(4) If $\mathcal{R}^{\mathcal{K}'}$ is an empty set, then $\mathcal{K} \not\models \alpha \mid\sim \beta$, else if $\mathcal{R}^{\mathcal{K}'}$ contains at least one rank, we return to (1) with $\mathcal{K}'$.

2.5 Lexicographic Closure

Lexicographic closure [13] is different from rational closure in that lexicographic closure follows presumptive reading whilst rational closure follows prototypical reading. Presumptive reading means that atypical formulas are able to inherit the properties of more typical formulas. Lexicographic closure shows this change by changing how rankings are formed. Lexicographic closure still uses the Base Rank Algorithm however formulas within the ranks are also ranked where the more typical formulas are ranked higher. By doing this, we can check and remove one statement within a rank instead of the whole rank. If all of the statements are removed then the whole rank is removed.

This method will be shown by giving the classic penguin example of entailment that compare rational closure to lexicographic closure. Given the knowledge base:

$$\mathcal{K} = \{b \mid\sim f, p \rightarrow b, b \mid\sim w, p \mid\sim \neg f\}$$

We can use the Base Rank Algorithm to create the following ranks:

∞	$p \rightarrow b$
1	$p \rightarrow \neg f$
0	$b \rightarrow f, b \rightarrow w$

Table 1: Ranks of $\mathcal{K}$

We then wish to query if $p \mid\sim w$ is entailed by the knowledge base:

2.5.1 Rational Closure Method.

(1) Query if $\overrightarrow{\mathcal{K}} \models \neg p$. $\neg p$ is entailed so we remove rank 0, which gives:

∞	$p \rightarrow b$
1	$p \rightarrow \neg f$
0	~~$b \rightarrow f, b \rightarrow w$~~

Table 2: Ranks of $\mathcal{K}$ After Rank 0 Removal

(2) We then query if $\neg p$ is entailed by the remaining statements in the ranks. $\neg p$ is not entailed and so we check if $\overrightarrow{\mathcal{K}}$ entails the statement $p \rightarrow w$. In this case it does not, therefore $\mathcal{K} \not\mid\approx p \mid\sim w$.

2.5.2 Lexicographic Closure Method.

(1) Rank each statement within the ranks based on all possible rankings. Here, only rank 0 changes.

∞	$p \rightarrow b$
1	$p \rightarrow \neg f$
0	$b \rightarrow f$
	$b \rightarrow w$

Table 3: Lexicographic Ranking 1

∞	$p \rightarrow b$
1	$p \rightarrow \neg f$
0	$b \rightarrow w$
	$b \rightarrow f$

Table 4: Lexicographic Ranking 2

(2) Query $\overrightarrow{\mathcal{K}} \models \neg p$. $\neg p$ is entailed so the top statement is removed from each ranking.

∞	$p \rightarrow b$
1	$p \rightarrow \neg f$
0	~~$b \rightarrow f$~~
	$b \rightarrow w$

Table 5: Lexicographic Ranking 1 After Removal

∞	$p \rightarrow b$
1	$p \rightarrow \neg f$
0	~~$b \rightarrow w$~~
	$b \rightarrow f$

Table 6: Lexicographic Ranking 2 After Removal

(3) We then query if the remaining statements entail $\neg p$. In this example, ranking 1 does not and so we check if the statements entail $p \rightarrow w$. The statements do entail $p \rightarrow w$ and so we conclude that $\mathcal{K} \mid\approx p \mid\sim w$.

This has demonstrated that presumptive reasoning is used for lexicographic closure since it was presumed that penguins have wings. Since rational closure uses prototypical reasoning, rational closure would have come to the opposite conclusion.

3 PROJECT AIMS

The key aims of this project are as follows:

- Develop a console application-based defeasible reasoner that reasons about some defeasible knowledge base by implementing the lexicographic closure algorithm.
- Refine the optimisation approaches used to increase the scalability of the lexicographic closure implementation compared to the previous year's SCADR project.
- Gather quantitative and qualitative information that shows which knowledge bases and queries the optimisations are most effective in obtaining better performance than previous implementations.

4 OPTIMISATION IMPLEMENTATIONS OF LEXICOGRAPHIC CLOSURE

The project introduces three optimisation implementations. the first is a more optimised iteration of the power set implementation from last year [18]. The second optimisation was the use of the Fibonacci search integrated with the ternary search implementation [18]. This optimisation will utilise the new power set implementation. This implementation will be compared to the binary and ternary implementations from last year [18]. The final optimisation uses multi-threading and concurrency to allow for multiple queries to be input and queried simultaneously.

The new implementations use the knowledge representation form of classical logic. The implementations take in a defeasible knowledge base and a defeasible query (queries for the concurrent algorithm) as inputs. The base rank algorithm will construct the rankings for the knowledge base. If there are classical statements, they will be added to the infinite rank. If not, the infinite rank remains empty. Defeasible statements are converted into classical statements and to their associated ranks based on the algorithm. The chosen implementation then checks whether the knowledge base entails the query. For this project, Java was used for development so that the *TweetyProject* [10] and Java Micro-bench Harness libraries could be integrated with the implementations.

To analyse these implementations, the time and space complexities will be examined. Time complexity measures the execution

time of an algorithm relative to the size of the input. For the time complexity, we will only take the number of executed entailment checks into consideration rather than the execution time of the satisfiability solver. This is because the paper only focuses on the implementations and not the satisfiability solver itself. Space complexity measures the space that is taken up by the algorithm relative to the input size. The Big O notation will be used to measure these complexities. This notation measures the upper bound of the complexity of the algorithm and disregards constants and lower order terms.

4.1 TweetyProject

TweetyProject [10] is a multi-faceted collection of Java libraries that is used for AI and knowledge representation and reasoning. This project specifically uses the Propositional Parser, Belief Revision, and Satisfiability Reasoner libraries from *TweetyProject*. The SAT solver is the most important aspect of the project since it determines whether the given query satisfies the knowledge base. For the SAT solver, the project uses the *Sat4jSolver* since *TweetyProject* has built-in support for it.

4.2 Power Set Optimisation

4.2.1 Analysis. This optimisation was first developed by Daniel Park [18] in the previous year. They had noticed that in lexicographic closure, the largest part of the algorithm is to prove if the negation of the antecedent of the query is used in a subset of statements in the worst rank. This typically means that each statement in the rank would have to be checked. However, by using the definition of lexicographic closure used for Datalog [17], the subsets can be combined into a single statement using disjunctions and conjunctions.

Definition 4.1. A power set of a set $\mathcal{S}$ is the set of all subsets of S. This includes set S itself and the empty set, denoted as $\mathcal{P}(\mathcal{S})$.

The previous implementation of the power set function used the previous definitions to first store the power set into a tuple of statements and then combine them using conjunctions and disjunctions separately. To combine them, each subset would be combined using conjunctions and the sets containing the same number of statements would be combined using disjunctions. This combination of statements does not include the empty set. For example, for a set $\mathcal{S} = \{a, b, c\}$, $\mathcal{P}(\mathcal{S}) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. This would then be combined to create:

$$\{\{a\} \vee \{b\} \vee \{c\}, \{a \wedge b\} \vee \{a \wedge c\} \vee \{b \wedge c\}, \{a \wedge b \wedge c\}\}$$

With the previous power set algorithm, the algorithm would first create the power set, store the power set, sort the power set, and then combine the power set using conjunctions and disjunctions. The prevPerm and printPowerSet algorithms improve upon this by combining these steps. The printPowerset algorithm creates an array that uses a binary indexing method where a 0 excludes the statement and 1 includes the statement in the subset. The prevPerm then checks that every permutation of that binary array has been used. This ensures that the power set is created in ascending order of number of statements in a subset. The printPowerset algorithm also adds the conjunctions and disjunctions while creating the power set. The subsets and the operators are added onto a single string before being stored. The algorithm will also remove the empty set and the subset that includes the whole set since these are redundant. As an example of these algorithms, let a rank $\mathcal{R}$ consist of statements $\{0 \rightarrow 1, 0 \rightarrow 2, 0 \rightarrow 3\}$. The printPowerSet($\mathcal{R}$, length of R) function will then return $\{\{0 \rightarrow 1\} \vee \{0 \rightarrow 2\} \vee \{0 \rightarrow 3\}, \{0 \rightarrow 1 \wedge 0 \rightarrow 2\} \vee \{0 \rightarrow 1 \wedge 0 \rightarrow 3\} \vee \{0 \rightarrow 2 \wedge 0 \rightarrow 3\}\}$.

Algorithm 2: prevPerm

Data: An integer array S
Result: **True**, if i > 0, and **False** otherwise

```
n := S.length − 1;
i := n;
while i > 0 and S_{i−1} ≤ S_i do
    i := i − 1;
end
if i ≤ 0 then
    return False;
end
j := i − 1;
while j + 1 ≤ n and S_{j+1} < S_{i−1} do
    j := j + 1;
end
temp := S_{i−1};
S_{i−1} := S_j;
S_j := temp;
size = n − i + 1;
for k = 0; k < floor(size/2); k = k + 1 do
    tem := S_{k+i};
    S_{k+i} := S_{n−k};
    S_{n−k} := tem;
end
return True;
```

These algorithms are then used by the lexicographic closure algorithm. This algorithm was originally created by Park [18] but there are new changes to this algorithm. The previous version used to output the incorrect answer when given certain queries that have the same formulas as the formulas in the knowledge base. The output should return true since the information is entailed by the knowledge base (it is the same information) but would instead return false. This would mainly occur with the first few ranks since the negation of the antecedent would not be entailed by the knowledge base. Therefore, the new algorithm will check the entailment of the formula if this occurs.

The below time and space complexities are for the worst-case scenario.

4.2.2 Time Complexity. The time complexity of the previous lexicographic closure algorithm (Algorithm 4) is O(n). This complexity remains the same since the main focus of the power set optimisation has been to reduce the space complexity. This complexity was broken down as follows:

Algorithm 3: printPowerSet

Data: An object array S_n, integer n
Result: A string arraylist *subs*
Create $subs_0, subs_1, \ldots, subs_n$;
answer ="(";
Create $contain_0, contain_1, \ldots, contain_n$;
$contain_0, contain_1, \ldots, contain_n := 0$;
for $i = 0;\ i < n;\ i = i + 1$ **do**
  $contain_i := 1$;
  Create $Con_0, Con_1, \ldots, Con_n$;
  for $index = 0;\ index < n;\ index = index + 1$ **do**
    $Con_{index} = contain_{index}$;
  end
  do
    for $j = 0;\ j < n;\ j = j + 1$ **do**
      if $Con_j = 1$ **then**
        $answer := answer + S_j.toString$+ " && ";
      end
    end
    $answer := answer.substring(0, answer.length - 4)$;
    $answer := answer$+ ") || (";
  while *prevPerm(Con)*;
  $answer := answer.substring(0, answer.length - 5)$;
  Add *answer* to *subs*;
  answer := "(";
end
Remove $subs_n$ from *subs*;
return *subs*;

Algorithm 4: Lexicographic closure using printPowerSet(*R*, length of *R*)

Data: A knowledge base $\mathcal{K}$ and a defeasible query $\alpha \mid\sim \beta$
Result: True, if $\mathcal{K} \mid\approx \alpha \mid\sim \beta$, and **False** otherwise
$(R_0, R_1, \ldots, R_{n-1}, R_\infty, n)$:= BaseRank($\mathcal{K}$);
$i := 0$;
$R := \bigcup_{i=0}^{k \leq n-1} R_k$;
while $R \neq \emptyset$ **do**
  if $R_\infty \cup R \models \neg\alpha$ **then**
    for $PS \in$ *printPowerSet*(R_i) **do**
      $R := R \backslash R_i$;
      $R := R \cup PS$;
      if $R_\infty \cup R \not\models \neg\alpha$ **then**
        if $R_\infty \cup R \models \alpha \rightarrow \beta$ **then**
          return $R_\infty \cup R \models \alpha \rightarrow \beta$;
        end
      end
    end
    $R := R \backslash R_i$;
    $i := i + 1$;
  else
    if $R_\infty \cup R \not\models \neg\alpha$ **then**
      return $R_\infty \cup R \models \alpha \rightarrow \beta$;
    end
  end
end
return $R_\infty \cup R \models \alpha \rightarrow \beta$;

- If the lowest rank contains n statements, then Algorithm 4 executes the entailment checker n times.
- If each rank consists of k statements, then Algorithm 4 will be called k times.

This would form a total of k x n. With $k < n$, the time complexity becomes O(n).

4.2.3 Space Complexity. For the space complexity, the previous implementations required O(2^n) since it stored the full power set [20] before combining the power set. For the new algorithms:

- The algorithm clones the whole knowledge base of size k.
- If the worst rank consists of n statements, then the algorithms stores $2n - 1$ statements at a time (the empty set is not created).

This would form a total of $2n - 1 + k$ which concludes to O(n) for space complexity.

4.3 Fibonacci Search Optimisation

4.3.1 Analysis. This implementation builds upon the work that was done for the power set optimisation. The Fibonacci search optimisation is a combination of the Fibonacci search algorithm and the ternary search optimisation from last year [18]. The typical ternary search algorithm calculates the two positions needed to divide a sorted array into thirds. Each third would then be evaluated to find the element requested. The algorithm will disregard the thirds that do not contain the element, continue with the remaining third, and repeat the process until the element is found.

However, for the ternary search optimisation, these positions are instead used as ranges to search a knowledge base. The SAT solver then uses these ranges to check for the negation of the antecedent. Given the positions *pos*1 and *pos*2 where $pos1 \leq pos2$, the algorithm would SAT solve from *pos*1 to the end of the knowledge base. If the negation of the antecedent is entailed, then the SAT solver checks from *pos*2 to the end. If this range is not entailed, the SAT solver checks from *pos*1 to *pos*2. If none of these ranges are entailed, then the process is repeated from the beginning of the knowledge base to *pos*2. This is done until the correct rank is found. When it is found, the lexicographic closure algorithm is used on the remaining statements and the query.

The typical Fibonacci search calculates the Fibonacci sequence until the final value goes over the size of the array. The last three values (e.g. 1 + 2 = 3 uses the values 1,2,3 for a size of 3) are used to generate a position of a sorted array. This position is then used to find the requested element in the array. If the element is less than the search request, the variables that store the Fibonacci values are decreased to the previous Fibonacci step (e.g. from 1 + 2 = 3 to 1 + 1 = 2) and the position is changed by these values and an offset. If the element is greater than the search request, then the Fibonacci values will be

changed in such a way that the new position will be above the old position.

The Fibonacci search optimisation combines these algorithms by using the generated Fibonacci numbers as the ranges mentioned in the ternary search optimisation. The algorithm first uses the size of the knowledge base to calculate the three values however, the third Fibonacci value is not used. The algorithm uses the first two values to act as $pos1$ and $pos2$ (e.g. 3 + 5 = 8 uses the values 3,5 for a size of 8). Once the algorithm starts repeating the whole process, the Fibonacci values are recalculated based on the size of the new range. This method was added to reduce the number of statements that the SAT solver would need to check at any one time. For example, for a knowledge base of 100 ranks, both $pos1$ and $pos2$ would initially be 33 for the ternary method while the Fibonacci method would be 55 and 89 respectively.

The below time and space complexities are for the worst-case scenario.

4.3.2 *Time Complexity.*

- If the worst rank consists of k statements, then the algorithm executes the SAT solver k times.
- To find the worst rank, where the subsets of the knowledge base have to be checked against the query, at least three entailment checks are required. It then takes $log_3(n)$ [1] times to find the rank.

This forms a total of $k + 3log_3(n)$ which leads to a time complexity of O($log(n)$).

4.3.3 Space Complexity. The ternary search implementation has a space complexity of O(2^n) due to the previous power set implementation. Since the Fibonacci search optimisation builds upon the new power set optimisation, the space complexity is reduced to O(n).

Algorithm 5: fibEntail

Data: $(R_0, R_1, \ldots, R_{n-1}, R_\infty, n), \alpha \mid\sim \beta, begin, end$
Result: $removeRank$

```
removeRank = −1;
size := end − begin;
fib2 := 0;
fib1 := 1;
fibM := fib2 + fib1;
while fibM < size do
    fib2 := fib1;
    fib1 := fibM;
    fibM := fib2 + fib1;
end
if begin < end then
    fib2 := fib2 + begin;
    fib1 := fib1 + begin;
    R := ⋃_{i=fib2+1}^{k≤n−1} R_k;
    if R_∞ ∪ R ⊨ ¬α then
        if fib1 < n then
            R := ⋃_{i=fib1+1}^{k≤n−1} R_k;
            if R_∞ ∪ R ⊨ ¬α then
                return fibEntail((R_0, R_1, ..., R_{n−1}, R_∞, n), α |∼ β, fib1 + 1, end);
            else
                R := ⋃_{i=fib1}^{k≤n−1} R_k;
                if R_∞ ∪ R ⊨ ¬α then
                    return fib1;
                else
                    return fibEntail((R_0, R_1, ..., R_{n−1}, R_∞, n), α |∼ β, fib2 + 1, fib − 1);
                end
            end
        else if fib1 = R.length then
            return fibEntail((R_0, R_1, ..., R_{n−1}, R_∞, n), α |∼ β, fib2 + 1, fib − 1);
    else
        R := ⋃_{i=fib2}^{k≤n−1} R_k;
        if R_∞ ∪ R ⊨ ¬α then
            return fib2;
        else
            return fibEntail((R_0, R_1, ..., R_{n−1}, R_∞, n), α |∼ β, begin, fib2);
        end
    end
else
    if begin = end then
        return begin;
    else
        return False;
    end
end
```

Algorithm 6: Lexicographic closure using Fibonacci search

Data: A knowledge base $\mathcal{K}$, a defeasible query $\alpha \mid\sim \beta$, begin, end
Result: **True**, if $\mathcal{K} \mid\approx \alpha \mid\sim \beta$, and **False** otherwise

$(R_0, R_1, \ldots, R_{n-1}, R_\infty, n) :=$ BaseRank($\mathcal{K}$);
$begin := 0$;
$end := n$;
$R := \bigcup_{i=0}^{k \leq n-1} R_k$;
$removeRank := fibEntail((R_0, R_1, \ldots, R_{n-1}, R_\infty, n), \alpha \mid\sim \beta, begin, end)$;
if $removeRank \neq \infty$ **then**
  $R := \bigcup_{i=removeRank}^{k \leq n-1} R_k$;
  for $PS \in printPowerSet(R_i)$ **do**
    $R := R \backslash R_i$;
    $R := R \cup PS$;
    if $R_\infty \cup R \not\models \neg\alpha$ **then**
      return $R_\infty \cup R \models \alpha \rightarrow \beta$;
    end
  end
  $R := R \backslash R_i$;
  return $R_\infty \cup R \models \alpha \rightarrow \beta$;
else
  return False;
end

4.4 Concurrent Optimisation

4.4.1 Analysis. So far the idea of scalability has been to reduce the execution time to evaluate one query with one knowledge base. The concurrent optimisation introduces a new interpretation of scalability. After the knowledge base has been generated, the optimisation takes in a text file of queries as input. The algorithm then creates three threads and separates the queries into three parts. It then calls the Fibonacci search implementation. The output will display the queries with their answer alongside them. This allows multiple statements to be done simultaneously instead of doing each statement sequentially. In terms of thread safety, the algorithm does not rely on shared memory and the algorithm waits until each thread has been completed before the entire application can continue.

4.4.2 Time Complexity. Since the concurrent optimisation creates threads to compute the Fibonacci search optimisation, the time complexity would still be O($log(n)$). Therefore, the speedup will be evaluated in the Results section.

4.4.3 Space Complexity. Since the concurrent optimisation creates threads to compute the Fibonacci search optimisation, the space complexity would be O(n).

4.5 Comparison of Complexities

It is clear that the space complexity of the new implementations has improved due to the power set implementation however, the time complexities have not changed. This is because the number of entailment checks have remained the same.

	Naive	Power Set	Binary Search	Ternary Search
Time	O(n^3)	$O(n)$	O($log(n)$)	O($log(n)$)
Space	O(n)	O(2^n)	O(2^n)	O(2^n)

Table 7: Time and Space Complexities of Previous Implementations

	Power Set	Fibonacci Search	Concurrent
Time	O(n)	O($log(n)$)	O($log(n)$)
Space	O(n)	O(n)	O(n)

Table 8: Time and Space Complexities of the New Implementations

5 EXPERIMENT DESIGN AND EXECUTION

This experiment aims to measure the performance of the implementations on knowledge bases with various number of statements, ranks, and distributions. The experiment will measure all of the implementations for lexicographic closure. This is because all of the implementations need to be measured from the same machine for comparable results. The test knowledge bases were made using the knowledge base generator by Bailey [2]. This generator allows for the statements, ranks, and distribution to be adjusted as parameters. The statements will be generated in such a way that the Base Rank algorithm will create the ranks that were requested. The statements will also be defeasible statements with numerical atoms e.g., $1 \rightarrow 0$. The current distributions of statements are uniform, normal, and inverse normal.

5.1 Specifications & Limitations

The machine used to test both the new and the old implementations are as follows:

- Model: Macbook Air (2015)
- CPU: Intel Core i5-5250U, 2 cores, 1.6GHz clock speed
- RAM: 2x4gb DDR3
- Power supply: 54 Whr

Due to the limitations of the previous power set implementation, the function cannot handle more than 13 statements within a single rank. This because the space complexity of the method is $O(2^n)$ which causes the machine to return "out of space" error. Since every implementation uses the Power Set implementation, the exponential and inverse exponential distribution cannot be tested and compared. Another limitation is that the concurrent implementation will use the System library instead of the Java Microbench Harness due to java heap space memory errors.

5.2 Hypothesis

This experiment has three main focuses. The first focus is comparing the new power set implementation to the previous implementation. Since the new version has a space complexity of O(n) while the previous version is O(2^n), the new version will be able to handle more statements. This change in space complexity will correlate with a better execution time since there is less time needed for reading memory.

The second is comparing the execution times of the binary search and ternary search with the Fibonacci search. From the theoretical analyses, binary search takes $2log_2(n)$ times, ternary search takes $3log_3(n)$ times, and Fibonacci search takes $3log_3(n)$. This shows that the Fibonacci search will perform better than the binary search. For the ternary search, the difference will be much smaller to notice. Although, the complexities are the same, the difference in execution time will be determined by how much information is being sent to the SAT solver. The Fibonacci search may require more look-ups but the reduction of time for the SAT solver will compensate for this. Therefore, the Fibonacci search will perform slightly better than ternary search.

The third focus is comparing the execution times of multiple queries on the Fibonacci search with the concurrent implementation. In this case, I expect the concurrent implementation to only perform better than the Fibonacci search. This is because creating multiple threads can cause allow more statements to be done in a shorter amount of time.

5.3 Java Microbench Harness

The Java Microbench Harness (JMH) is a tool that builds, runs, and analyses benchmarks for Java programs. JMH is part of the OpenJDK collection of libraries. JMH was chosen since it has many advantages/features that are better suited for testing. Firstly, results are not skewed from optimisations from the virtual machine. Secondly, any unused or useless code is removed before running. Lastly, it is also more accurate than using the System library's time functions. JMH will be used to test the execution time of the implementations around the SAT solver.

The tool will set up the knowledge base and call the Base Rank algorithm before measuring the execution time. This was done to not measure the creation of the knowledge base and to reduce any other external factors, thereby giving a more accurate measurement of the execution times of the algorithms.

JMH will use the following parameters:

- Fork (value = 2)
- Measurement (iterations = 10, time = 1)
- Warmup (iterations = 5, time = 1)

Fork refers to the number of trials that will be performed. Each trial consists of a set of warmups and iterations. Warmups refers to the number of warmup tests which are done to properly optimise the JVM before running the benchmark. This ensures that the JVM does not skew the benchmark results. The measurements refer to the number of times that the tests are run for each fork. This tracks the best, average, and worst times of each iteration. In total, the JMH will generate results from 20 runs.

5.4 Test Cases

Five test cases will be conducted for this experiment that use various knowledge bases. Test 1 will test the execution time of the ternary search, binary search, and Fibonacci search implementations. The knowledge base consists of 250 ranks with 1000 statements, containing only defeasible statements, and an uniform distribution. The query set for test 1 will use a statement from rank 0 and every 10^{th} rank (26 queries).

Test 2 will test the ternary search, binary search, and Fibonacci search implementations against all distributions (uniform, normal, inverted-normal). The knowledge base consists of 50 ranks with 200 statements, and only containing defeasible statements. The query set for test 2 will use a statement from every 7^{th} rank starting from rank 0 (8 queries).

Test 3 will compare the number of comparisons and execution times between the Fibonacci search, binary search, and ternary search. The results will also be used to compare the time complexities with the comparisons. The knowledge base consists of 50 ranks with 200 statements, containing only defeasible statements, and a uniform distribution. The comparisons will be measured using internal counters that count entailment checks. The query set for test 3 will use a statement from every 7^{th} rank starting from rank 0 (8 queries). The execution times will show the total time taken to run all queries.

Test 4 will compare the concurrent the execution time of the concurrent implementation with the Fibonacci search. The knowledge base will consist of 50 ranks with 200 statements, containing only defeasible statements, and a uniform distribution. The query set for test 4 will use a statement from every 7^{th} rank starting from rank 0 (8 queries). The execution times will show the total time taken to run all queries.

Test 5 will be used to test the execution time of the power set implementations. The knowledge base consists of 50 ranks with 200 statements, containing only defeasible statements, and an uniform distribution.The query set for test 5 will use a statement from every 2^{nd} rank starting from rank 0 (25 queries). The execution times will show the total time taken to run all queries.

5.5 Results

5.5.1 Test 1. The summary below and Figure 1 (See Supplementary Information (SI) A) shows that the Fibonacci search has the highest average time between the three implementations. This is due to the larger execution time in the second third of the knowledge base (between rank 80 and 160). This occurred due to the additional entailment checks required to reach that middle portion which in turn increased the average execution time. The Fibonacci search has, however, outperformed both implementations for the best case (located in rank 180). This is because the other implementations would require more entailment checks that check a large portion of the knowledge base to reach that rank.

	Binary Search	Ternary Search	Fibonacci Search
Best	324.804068	362.314401	294.660394
Average	513.9112347	544.1770445	554.1821034
Worst	884.727327	751.142284	769.239586

Table 9: Summary of Test 1 (in ms)

5.5.2 Test 2. The tables below and the figures in SI B have shown that the Fibonacci search has outperformed in most categories. This is particularly true for the normal and inverse normal distributions. This shows that the power set implementations have reduced the execution time through the space complexity. This is true since the ranks that contain the most statements are still much lower than that of the previous implementations.

The uniform distribution of Test 2 is in stark contrast of Test 1's uniform distribution. As the knowledge base has increased, the overhead of comparisons increase for ranks that are in the first two thirds while decreasing for the last third.

	Binary Search	Ternary Search	Fibonacci Search
Best	267.962712	204.036052	219.908109
Average	482.304789	386.6634044	384.9115139
Worst	754.731928	528.448789	534.819639

Table 10: Summary of Uniform Distribution for Test 2 (in ms)

	Binary Search	Ternary Search	Fibonacci Search
Best	252.424176	201.800873	195.10106
Average	447.3732729	385.905811	340.0518308
Worst	545.395394	526.146552	492.056261

Table 11: Summary of Normal Distribution for Test 2 (in ms)

	Binary Search	Ternary Search	Fibonacci Search
Best	320.194511	442.844887	218.072917
Average	483.9600438	840.2043348	452.3762614
Worst	653.11928	1033.944199	680.732539

Table 12: Summary of Inverse Normal Distribution for Test 2 (in ms)

5.5.3 Test 3. Table 13, Table 14, and Figure 5 (SI C) follow a similar trend to Test 2 in terms of execution time. In this case, the Fibonacci search implementation outperforms all categories for execution time. In terms of the comparisons, the Fibonacci search also outperforms in all categories. This is all due to the overall size of the knowledge base, how the positions ($pos1$ and $pos2$) are distributed, and where the rank is situated. There is also a positive correlation between execution time and comparisons. From Test 2's uniform distribution we can also see a correlation between the distribution and the number of comparisons required.

In terms of the worst-case scenarios, the implementations' comparisons match the time complexities as expected. The binary search is $2log_2(50) = 11.28$ which can be approximated to 13 (or at least higher than the other implementations). $3log_3(50) = 10.68 \approx 11$.

	Binary Search	Ternary Search	Fibonacci Search
Best	2974.545262	2486.428984	2637.942748
Average	3153.764501	2703.557163	2654.605634
Worst	3332.983739	2920.685342	2671.26852

Table 13: Summary of Execution Time for Test 3 (in ms)

	Binary Search	Ternary Search	Fibonacci Search
Best	6	4	3
Average	9.75	8.625	7.125
Worst	13	11	11

Table 14: Summary of Comparisons for Test 3 (in ms)

5.5.4 Test 4. The Fibonacci search without the concurrent implementation yielded an average time of 21999.35418 ms whilst the concurrent implementation yielded an average time of 6672.516258 ms. This leads to a speed up of 3.297010203 times.

5.5.5 Test 5. From the data below and Figure 6 (SI E), the new power set implementation is very similar in execution time and only slightly outperforms the previous power set method in terms of average time. This is due to the time complexity being very similar. The new power set also outperforms for the worst case execution time. The average time and worst case time can be attributed to the better space complexity as this affects the execution time. Although the old power set method has a better best case execution time, this is due to the implementation giving the incorrect answer for rank 0. Therefore, this best time must be disregarded, making the new power set method better by default.

	New Power Set	Old Power Set
Best	166.126	65.002704
Average	2478.15836	2480.408163
Worst	4898.107	5630.298724

Table 15: Summary of Test 5 (in ms)

5.6 Conclusions

Test 1 has shown that for larger knowledge bases, the overhead of execution times for the Fibonacci search implementation increases because the first two thirds become larger and therefore require more entailment checks.

Test 2 further proves that the Fibonacci search implementation is better for smaller knowledge bases unlike the binary search and the ternary search which improved as the size increased. The normal and inverse normal distributions also shows that the Fibonacci search implementation is better at handling ranks that had a large number of statements.

Test 3 has shown that the distribution of the statements, size of the knowledge base, and the number of ranks greatly affects how many comparisons are made. The test has shown that the distribution can also affect the number of comparisons. The test has also shown that the time complexities of the implementations closely match the comparisons which solidifies the time complexities.

Test 4 showed a large improvement for reading in and evaluating multiple queries at a time.
Test 5 has shown the importance of space complexity and how it affects execution times. It has also highlighted the issue within the previous power set implementation where the incorrect output is given for antecedents in the beginning ranks.

This paper has shown that the Fibonacci search optimisation did not prove to be more scalable than the ternary search or the binary search due to the increase in execution times for larger knowledge bases. However, by removing this particular method as a potential optimisation for defeasible reasoners, it has shed more light on how to build better scalable defeasible reasoners. It has also shown that the concurrent implementation is very effective in dealing with multiple queries at once and proves that scalability can be improved in more ways than one.

6 RELATED WORK

6.1 Classical SAT Solving

Rational closure and lexicographic closure reduce to a number of classical entailment checks. This makes Boolean satisfiability applicable to this research. Boolean satisfiability (SAT) determines whether there exists an assignment satisfying a given Boolean formula [[6], [19]]. In analyzing the SAT solvers for classical reasoning we can determine which SAT solvers are appropriate for our defeasible reasoning model.

6.1.1 Semantic Tableaux. The tableau decomposes a formula into sets of atomic literals, creating in a tree-like tableau [16]. Each branch ends with a complementary pair of formulae, a closed branch, or consists of a set of non-contradictory literals, an open branch. A model for the given formula is represented by an open branch. When decomposition is no longer possible, the construction is complete. If there is a clash then the initial formula is considered unsatisfiable. A clash this represents a contradiction and occurs when literals for an atom are found within the same subset. A completed tableau is required to determine a formula's satisfiability.

6.1.2 DPLL. DPLL is a backtracking algorithm [7]. The algorithm allows for the input of a propositional formula in CNF format. If the formula is satisfiable, the algorithm returns true. If the formula is unsatisfiable, the algorithm returns false. A branching procedure is executed to set some atom in α to a random truth value. Branching repeats until an assignment either satisfies α or not. If not, the algorithm will backtrack. Backtracking retrieves the most recent branching assignment and re-branches it with a different assignment. Backtracking will repeat if there are no new assignments. α is unsatisfiable if there exists no new branches and backtracking is not possible.

6.1.3 Conflict-Driven Clause Learning (CDCL). The creation of the CDCL [15] was motivated by the DPLL SAT solver. The distinction between the SAT solvers is that the CDCL SAT solver uses non-chronological backtracking [7]. Conflicts caused by variable assignments are cached. This leads to optimised efficiency and execution.

6.1.4 Defeasible SAT Solvers. SAT solvers only cover a limited scope of reasoning, specifically classical. SAT solvers currently cannot deduce the satisfiability of a defeasible statement in relation to the rational closure or lexicographic closure of a defeasible knowledge base [13].

6.2 SCADR Project (2021)

The Scalable Defeasible Reasoning research is work that is built upon last year's (2021) work and so is related to this project as well.

6.2.1 Park [18]. Daniel Park developed the lexicographic closure implementations which were used for comparison. He specifically developed the power set optimisation, the binary search optimisation, and the ternary search optimisation. He has also developed a concurrent base ranking model which was used for this project.

6.2.2 Rational Closure [9]. Although the work of Hamilton [9] was not directly about rational closure, theory and examples were discussed to understand lexicographic closure. Hamilton developed the rational closure implementations. This includes the binary search optimisation, an indexing optimisation, and a binary indexing optimisation.

6.2.3 Bailey [2]. A knowledge base generator had to be used for this project. This generator allowed for different parameters to be set such as number of ranks, number of statements, distributions, and optional inclusion of classical statements.

7 FUTURE WORK

The improvements made for this project focused on the implementations of lexicographic closure. Future researchers should consider improving upon the knowledge base generator created by Bailey. By generating more complicated statements, more tests and analyses can be done for the current and previous implementations. Although this paper did not focus on the Base Rank method, no changes were made to the previous implementations. Improving this algorithm would make ranking the knowledge base faster for larger data sets. More research should also be done for the concurrent implementation. With better thread creation and management, the implementation could become faster.

REFERENCES

[1] Nitin Arora, Mamta Martolia Arora, and Esha Arora. 2016. A Novel Ternary Search Algorithm. *International Journal of Computer Applications* 144, 11 (2016).

[2] Aiden Bailey. 2021. Scalable Defeasible Reasoning. (2021), 1–16. https://projects.cs.uct.ac.za/honsproj/cgi-bin/view/2021/bailey_hamilton_park.zip/files/BLYAID001_SCADR.pdf

[3] Giovanni Casini, Thomas Meyer, Kodylan Moodley, and Ivan Varzinczak. 2013. Towards practical defeasible reasoning for description logics. (2013), 1–13. http://ceur-ws.org/Vol-1014/paper_17.pdf

[4] Giovanni Casini, Thomas Meyer, and Ivan Varzinczak. 2018. Defeasible Entailment: from Rational Closure to Lexicographic Closure and Beyond. *In 17th International Workshop on Non-Monotonic Reasoning (NMR)* (2018), 109–118. http://pubs.cs.uct.ac.za/id/eprint/1304

[5] Giovanni Casini, Thomas Meyer, and Ivan Varzinczak. 2019. Taking defeasible entailment beyond rational closure. *European Conference on Logics in Artificial Intelligence*, 182–197. https://doi.org/10.1007/978-3-030-19570-0

[6] Stephen A. Cook. 1971. The complexity of theorem-proving procedures. *In Proceedings of the third annual ACM symposium on Theory of computing* (1971), 151–158. https://doi.org/10.1145/800157.805047

[7] Martin Davis, George Logemann, and Donald Loveland. 1962. A Machine Program for Theorem-Proving. *Commun. ACM* 5 (1962), 394–397. Issue 7. https://doi.org/10.1145/368273.368557

[8] L. Giordano, V. Gliozzi, N. Olivetti, and G.L. Pozzato. 2015. Semantic characterization of rational closure: From propositional logic to description logics, Artificial Intelligence. 226 (2015), 1–33. https://doi.org/10.1016/j.artint.2015.05.001

[9] Joel Hamilton. 2021. An Investigation into the Scalability of Rational Closure. (2021), 6–7. https://projects.cs.uct.ac.za/honsproj/cgi-bin/view/2021/bailey_hamilton_park.zip/files/SCADR_HMLJOE001.pdf

[10] FernUniversität in Hagen. 2022. TweetyProject - Home. https://tweetyproject.org/

[11] Adam Kaliski. 2020. An Overview of KLM-Style Defeasible Entailment. (2020), 52–83. http://hdl.handle.net/11427/32743

[12] Sarit Kraus, Daniel Lehmann, and Menachem Magidor. 1990. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial intelligence* 44 (1990), 167–207. https://doi.org/10.1016/0004-3702(90)90101-5

[13] Daniel Lehmann. 1995. Another perspective on default reasoning. *Annals of Mathematics and Artificial Intelligence* 15, 1 (1995), 61–82. https://doi.org/10.48550/arXiv.cs/0203002

[14] Daniel Lehmann and Menachem Magidor. 1992. What does a conditional knowledge base entail? *Journal of Artificial Intelligence* 55, 1 (1992), 1–60. https://doi.org/10.1016/0004-3702(92)90041-U

[15] Joaäo P. Marques-silva and Karem A. Sakallah. 1999. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.* 48 (1999), 506–521. https://doi.org/10.1109/12.769433

[16] Ben-Ari Mordechai. 2012. *Propositional Logic: Formulas, Models, Tableaux* (3 ed.). Springer London, 1, 7–47. https://doi.org/10.1007/978-1-4471-4129-7

[17] Matthew Morris, Tala Ross, and Thomas Meyer. 2020. Algorithmic definitions for KLM-style defeasible disjunctive Datalog. *South African Computer Journal* 32, 2 (2020), 141–160. https://doi.org/10.18489/sacj.v32i2.846

[18] Joon Soo Park. 2021. Scalable Defeasible Reasoning. (2021), 1–12. https://projects.cs.uct.ac.za/honsproj/cgi-bin/view/2021/bailey_hamilton_park.zip/files/PRKJOO001_SCADR_Final_Paper.pdf

[19] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. 2015. Boolean Satisfiability Solvers and Their Applications in Model Checking. *Proc. IEEE* 103 (2015), 2021–2035. https://doi.org/10.1109/JPROC.2015.2455034d

[20] Cong-Cong Xing. 2016. Seven different proofs for$|\mathcal{P}(A)| = 2^n$. *Journal of Computing Sciences in Colleges* 31, 5 (2016), 53–61.

Dhiresh Thakor Vallabh

Supplementary Information

A TEST 1

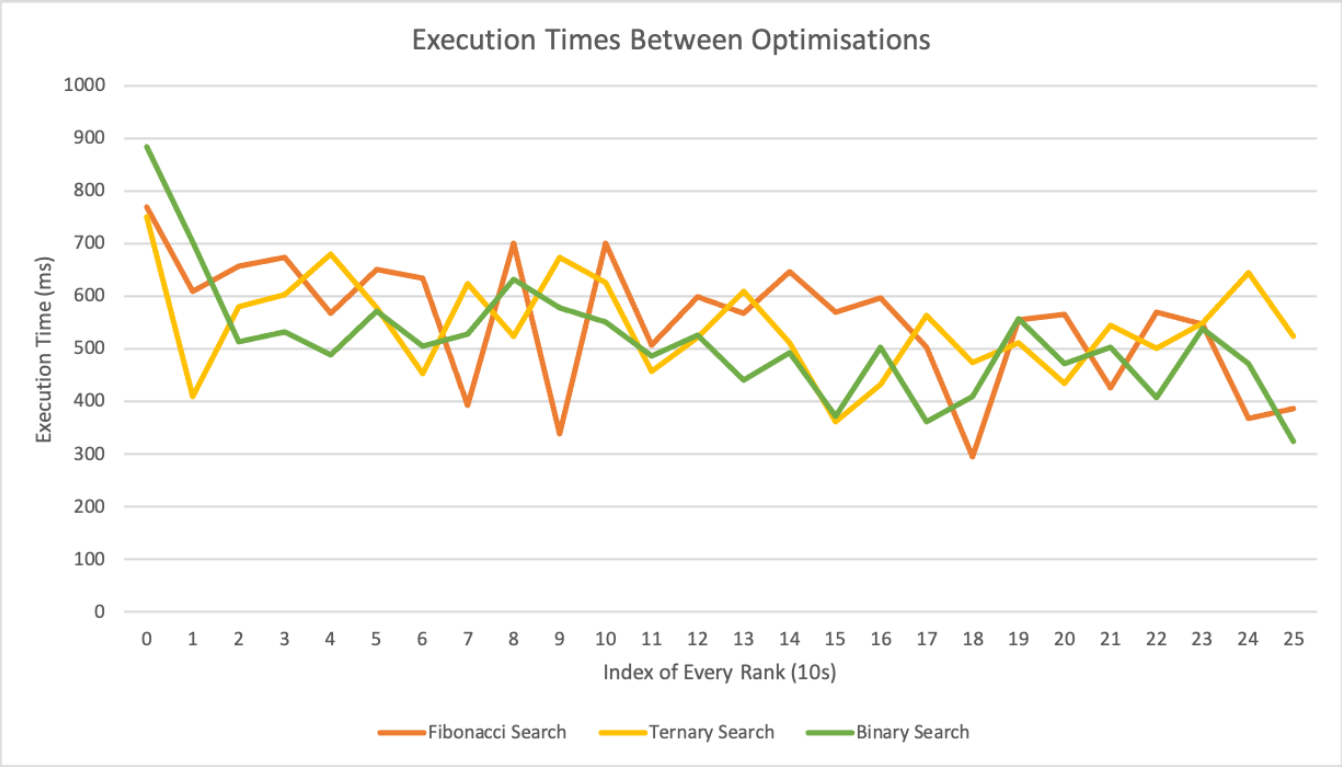

Figure 1: Comparison of Execution Times for Implementations Using Fibonacci Search, Ternary Search, and Binary Search

B TEST 2

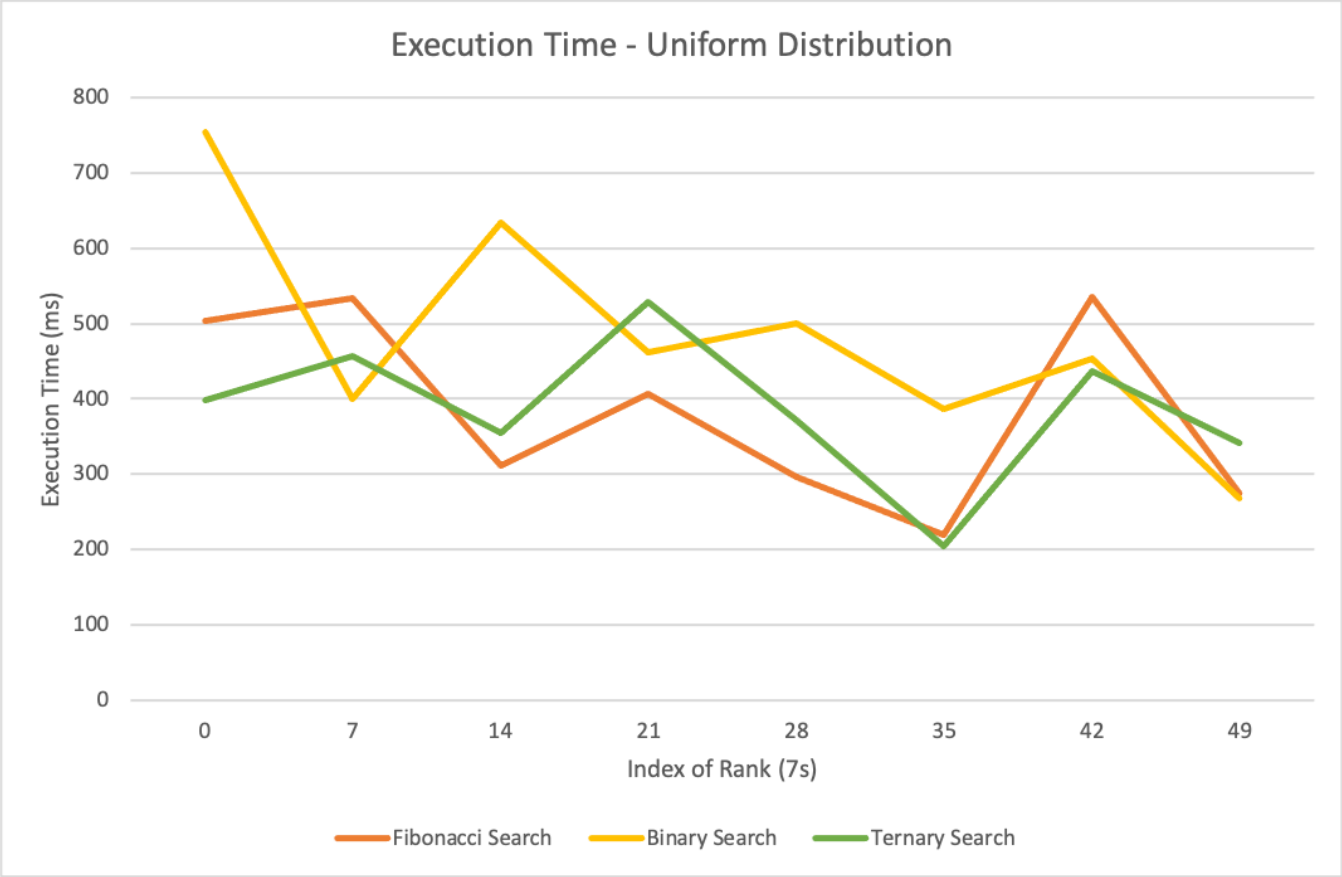

Figure 2: Comparison of Execution Times for Implementations Using Fibonacci Search, Ternary Search, and Binary Search - Uniform Distribution

Figure 3: Comparison of Execution Times for Implementations Using Fibonacci Search, Ternary Search, and Binary Search - Normal Distribution

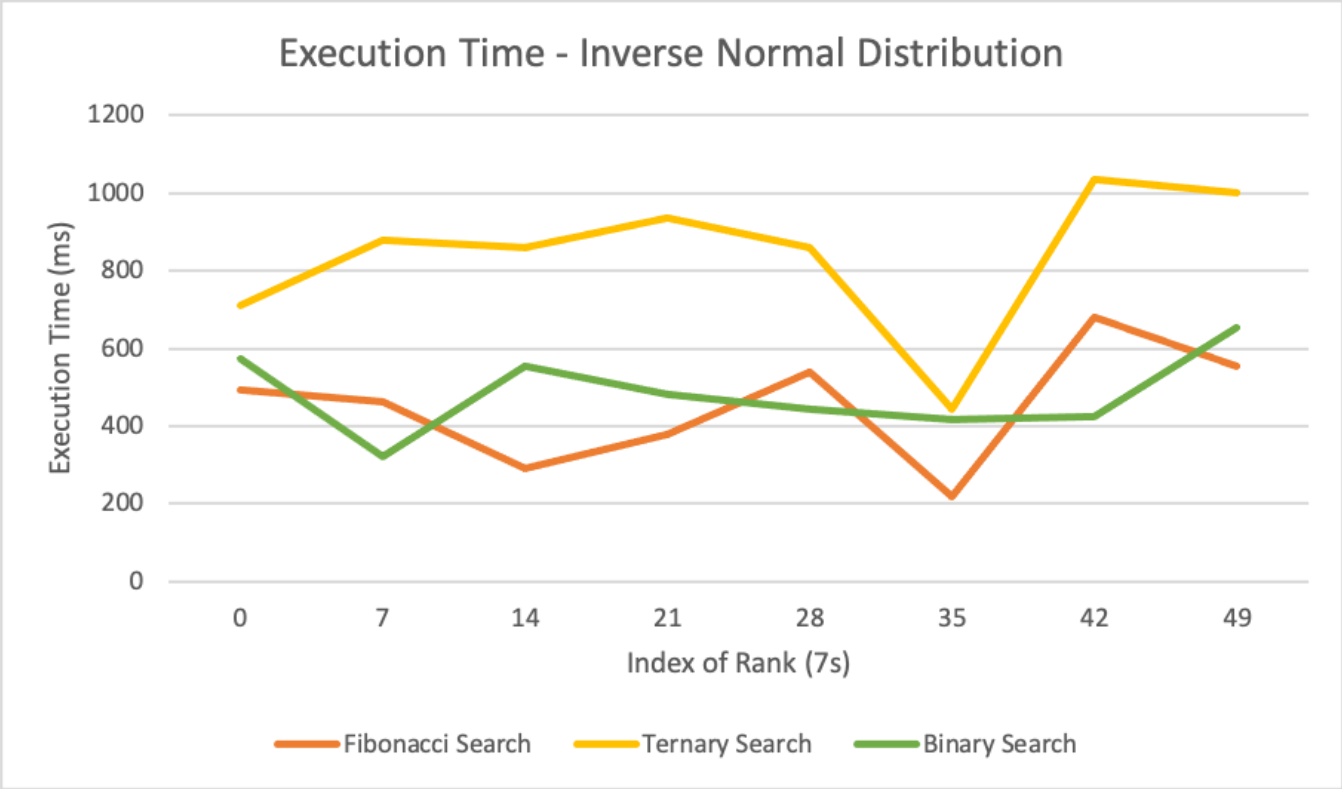

Figure 4: Comparison of Execution Times for Implementations Using Fibonacci Search, Ternary Search, and Binary Search - Inverse Normal Distribution

C TEST 3

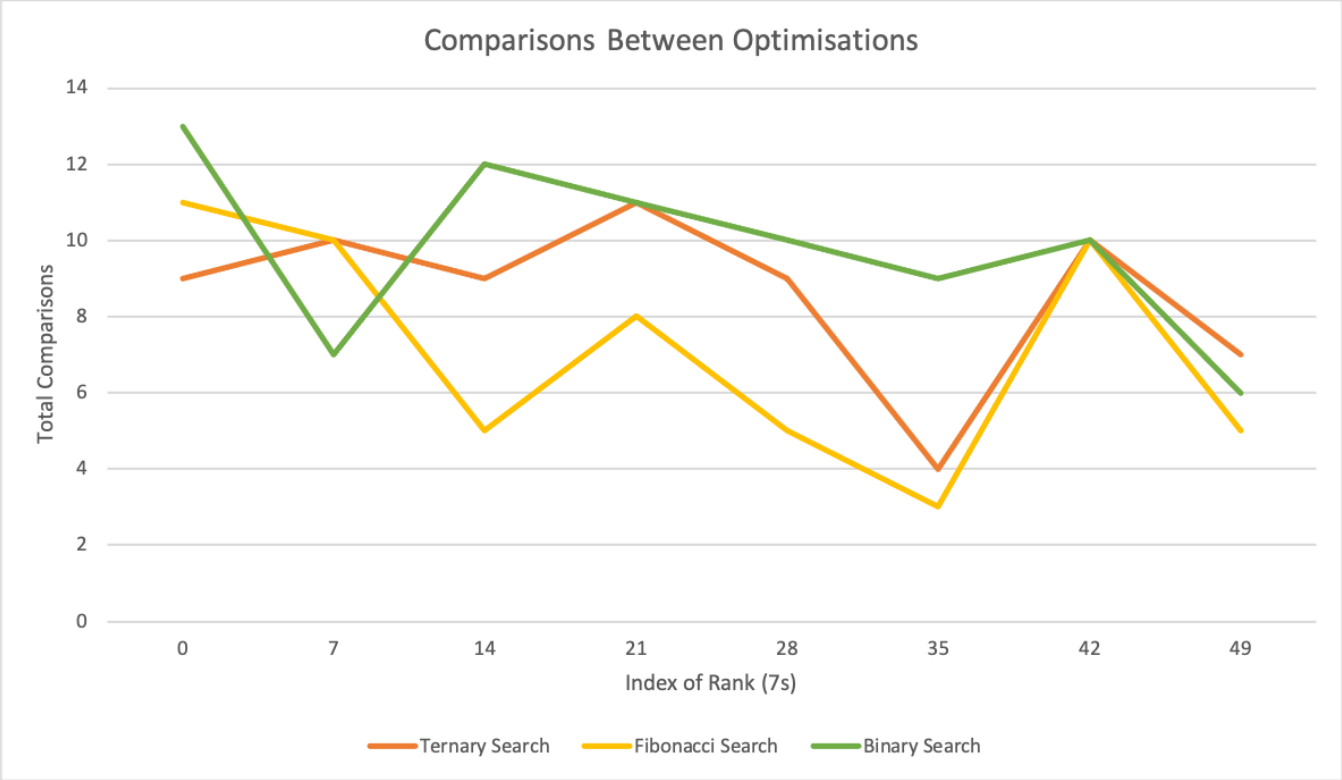

Figure 5: Comparison of Entailment Checks for Implementations Using Fibonacci Search, Ternary Search, and Binary Search

D TEST 5

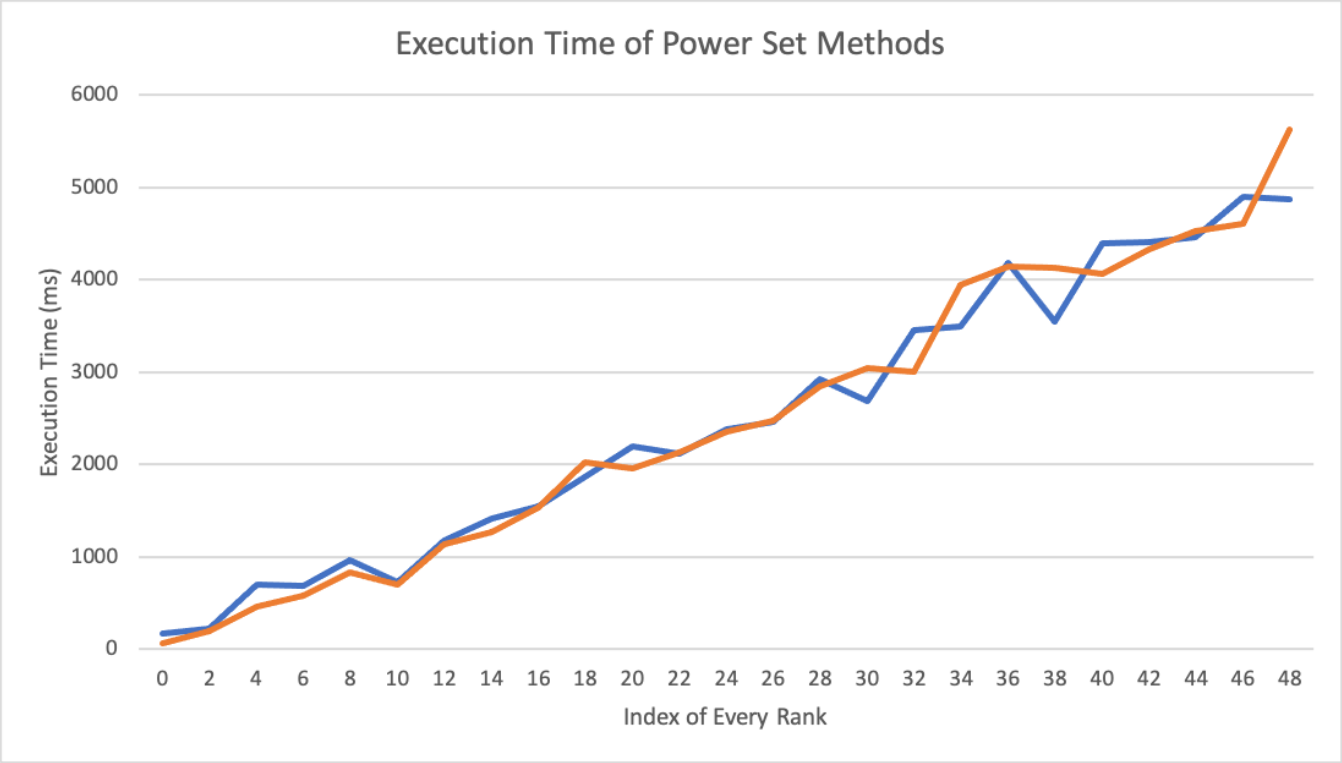

Figure 6: Comparison of Execution Times for the New and Old Power Set Implementations