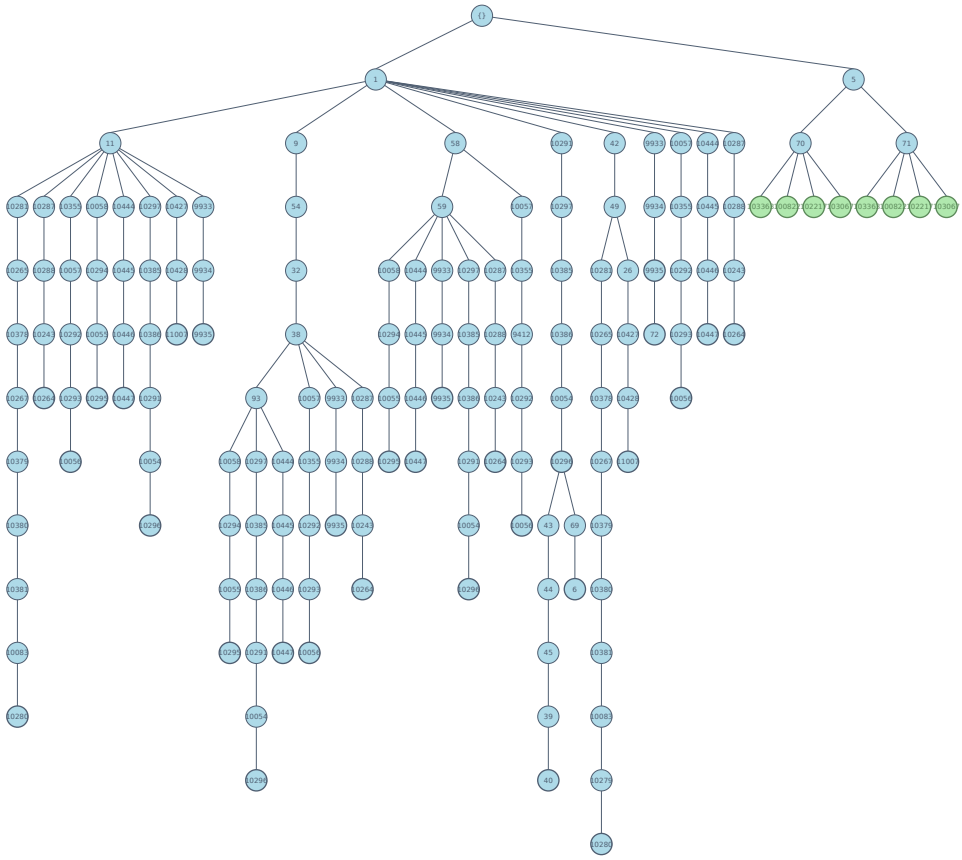
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Weighted Set Containment in Redundant Data Sets

Master's thesis in Computer Science: Algorithms, Languages & Logic

JOHAN WERMENSJÖ
JOHAN GRÖNVALL

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

Weighted Set Containment in Redundant Data Sets

JOHAN WERMENSJÖ
JOHAN GRÖNVALL

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Weighted Set Containment in Redundant Data Sets
JOHAN WERMENSJÖ
JOHAN GRÖNVALL

© JOHAN WERMENSJÖ, 2017.
© JOHAN GRÖNVALL, 2017.

Supervisor: Peter Damschke, Computer Science and Engineering
Advisor: Oscar Bäckström, Volvo Group IT
Examiner: Devdatt Dubhashi, Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 10 00

Cover: A set-trie constructed by the greedy algorithm proposed in this thesis.

Typeset in LaTeX
Gothenburg, Sweden 2017

Weighted Set Containment in Redundant Data Sets
JOHAN WERMENSJÖ
JOHAN GRÖNVALL
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Given a set family F and a query set Q, *weighted set containment* is the problem of finding the set $C \in F$ with the largest sum of weights, such that $C \subseteq Q$, where every element has a corresponding weight. This problem was investigated at the request of Volvo Group IT, who rely heavily on weighted set containment queries in many of their applications. In this thesis we show that weighted set containment can be solved efficiently using trie based preprocessing when applied to redundant data sets. We show that finding the most efficient trie which represents F is NP-complete and we introduce a number of approximation algorithms. We show through empirical testing that some of our algorithms outperform state-of-the-art methods for similar problems when applied to Volvo's particular data set.

Keywords: trie, set-trie, set containment, weighted set containment.

Acknowledgements

We would like to thank Volvo Group IT for the opportunity to work with them and our supervisor at Volvo Oscar Bäckström for giving us the freedom to take the project in the direction we wanted. We would also like to thank our supervisor at Chalmers, Professor Peter Damaschke for his advice and in particular his idea to show NP-completeness of the set-trie problem by reduction from vertex cover.

Johan Wermensjö, Gothenburg, June 2017
Johan Grönvall, Gothenburg, June 2017

Contents

List of Figures

List of Tables

1

Introduction

Set containment is the problem of determining whether a set is a subset of another set. It is a natural and ubiquitous problem found in many areas of Computer Science. This thesis deals with what we call weighted set containment, which is used in an algorithm at Volvo Group IT to find relevant information for a specific vehicle. The primary purpose of this thesis is to optimize this algorithm and to investigate how preprocessing can be utilized to improve the performance of weighted set containment queries.

The algorithm is used in many different applications at Volvo, some of which are limited by the performance of the current algorithm. In order to compensate, some applications precompute the results for vast amounts of possible inputs to maintain acceptable performance. If a significantly faster method can be developed, these limitations can be lifted and precomputation can perhaps be avoided entirely. In addition to the clear benefits for Volvo, since the problem is quite natural and can be applied to many areas, theoretical research regarding the problem could prove useful in many different domains.

The current algorithm at Volvo outputs the best fitting descriptions, given a description of a vehicle and a set of descriptions that map information to vehicles. The descriptions are modeled as sets of attributes and the best fitting information description(s) are the one(s) with the highest score. The score is calculated by giving a point for every attribute that exists both in the vehicle description and in the information description. If an attribute is present in the information description but not in the vehicle description, then the information description is disqualified i.e. it is not considered a valid output. There are also certain attributes which can disqualify but give no points.

In a more general sense, the problem can be seen as computing the highest scoring set(s) among a family of candidate sets $F = \{C_1, C_2, ..., C_n\}$. A set is only valid if it is a subset of the so called query set Q. Every element in either a candidate set or the query set corresponds to an attribute. Since not all attributes award the same score, each element x_i is associated with a corresponding weight $w(x_i)$.

The problem is similar to set containment join, see section 2.2, which is a basic operator in all modern data management systems and has a wide range of applications [1]. Weighted set containment can be solved by first running a set containment join, and then linearly computing the highest scoring set from the returned pairs. The state of the art solution for solving set containment join involves adaptively creating

a prefix tree and an inverted index from the input data during the runtime of the algorithm [1]. This solution would be performing a lot of unnecessary computations, even if it were to use the current state of the art algorithm for set containment join. This is because set containment join solves a more general problem.

The similarities between the two problems suggest that weighted set containment can be solved efficiently using similar data structures. Furthermore, having these data structures already preprocessed as opposed to adaptively created, could prove very efficient.

1.1 Aim

The primary goal of the project is to reduce the execution time of Volvo's current algorithm. Additionally, it aims to investigate whether preprocessing can be utilized to improve the search strategy.

Although the problem is simpler than set containment join and can be solved by a linear search strategy, there are still several potential areas of improvement, especially given the opportunity to preprocess the data. Firstly, the set containment problem could be solved with trie structures [2]. Secondly, smarter search strategies that realize earlier on that the highest scoring candidate has been found, could be implemented. Finally, data points that do not change the outcome of the search could be removed in the preprocessing step.

1.2 Scope

This thesis presents and compares methods based on the particular data set at Volvo Group IT, other data sets could end up with very different results. Implementation of the different strategies are done using Java and SQL, and no other languages or tools are considered. This is because it is mainly the relative performance that is relevant. The chosen languages are therefore, for simplicity, the same as the ones used by Volvo. Furthermore the thesis does not consider the performance of different database or storage technologies, but rather focuses on the algorithm itself.

The algorithms are primarily evaluated through empirical testing and the algorithm currently in use at Volvo serves as a baseline for comparison. All tests are based on a subset of the actual data set, due to the large amount of data.

1.3 Problem Definition

Let F be a family of n candidate sets $F = \{C_1, C_2, ..., C_n\}$. Denote the cardinality of each candidate set with m_i such that $\forall C_i \in F, m_i = |C_i|$ and denote $m = max(\{|C_i| : C_i \in F\})$. Let $U = \bigcup_{C_i \in F} C_i$ and let each element $x_i \in U$ have a corresponding weight $w(x_i)$ where $w : U \to \mathbb{R}_+$. The sum of weights of any candidate set $C_j, \sum_{x_i \in C_j} w(x_i)$ is referred to as its score. We define the weighted set containment problem as finding the candidate sets C_j with the highest score such that $C_j \subseteq Q$, where Q is the query set.

2

Theory

This chapter describes topics relevant to weighted set containment. Additionally, the general optimization method simulated annealing is briefly described. Finally, the problem areas that affect the practical evaluation of our proposed methods are elaborated upon.

2.1 Trie

A trie is a search tree used for storing sets of sequence data or for representing an associative array, where the key is usually a string [3]. The main advantages with tries is their fast retrieval time, easy updating, convenience in handling arguments of diverse lengths and their ability to take advantage of redundancies in the stored data [4].

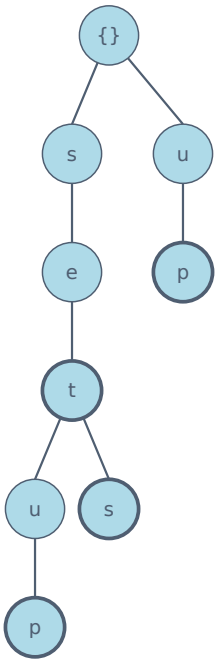


Figure 2.1: Shows a trie containing the strings ”set”, ”setup”, ”sets”, and ”up”. Nodes represent characters and end nodes are indicated by a thicker outline.

When used for strings, every node in a trie represents a character and every path from the root node to an end node represents a stored word. Figure 2.1 shows a trie, notice that the commonly occurring substring "set" is only stored once. This makes tries an efficient data structure for storing redundant data. However, the data structure is unable to take advantage of redundancies that occur later in the string, if it is prefixed by a unique substring. For example, the substring ”up” in figure 2.1 occurs twice in the set but is not compressed since it in the second case is prefixed by ”set”.

2.1.1 Set-trie

The name ”set-trie” is used by Savnik [2] to describe a trie that stores sets rather than sequence data. In contrast to traditional tries, the weighted set containment problem demands no particular order of the elements and therefore every order is a valid option. To build a trie from a family of sets, Savnik assigned an arbitrary order to the elements and inserted every element in this sorted order. However, the amount of data that is compressed depends strongly on the order in which the elements are inserted into the trie, making this approach naive. Although Savnik does not introduce any particularly efficient ways of constructing the tries, the name set-trie is descriptive and well suited for our problem and is therefore used in the thesis.

2.2 Set Containment Join

A problem closely related to weighted set containment which has been thoroughly studied, is the set containment join problem. Set containment join $R \bowtie_{\subseteq} S$ takes as input two collections of sets R and S and returns all pairs $(r, s) \in R \times S$ where $r \subseteq s$. For example, if $R = \{\{2\}, \{1, 2\}, \{2, 3, 4\}\}$ and $S = \{\{1, 2, 3\}, \{2, 3, 4, 5\}\}$, then $R \bowtie_{\subseteq} S$ would give $(\{1, 2\}, \{1, 2, 3\})$, $(\{2, 3, 4\}, \{2, 3, 4, 5\})$, $(\{2\}, \{1, 2, 3\})$, $(\{2\}, \{2, 3, 4, 5\})$.

An efficient algorithm for performing set containment join was developed by Jampani et al. [5]. The algorithm uses an inverted index to efficiently find all the records that contain a given set. In addition, prefix trees are used to efficiently mine frequent itemsets, and to help avoid redundant work. Note that a prefix tree is essentially a trie where only the prefix of any sequence is stored and therefore, just like a trie, any set inserted into the tree must be assigned an order. Jampani et al. assigned a descending global order to their elements based on how frequently they occurred in a given relation in an attempt to get a good compression.

Bouros et al. later improved upon the algorithm using an adaptive cost-model allowing them to only partly construct the prefix tree [1]. They also showed that they could get better performance by ordering their elements in the ascending order of frequency rather than descending, since such an ordering resulted in earlier disqualifications.

Note that the problem addressed in this thesis, weighted set containment, can be solved by first performing a set containment join , where $R = F$ and $S = \{Q\}$ and then linearly selecting the highest scoring elements from the returned pairs. However this would be inefficient since set containment join algorithms do not utilize the fact that we only have a single query set, i.e. $|S| = 1$. Furthermore, finding the highest scoring would be done after all the joins are performed, which could potentially be done during the joins.

2.3 Simulated Annealing

Simulated annealing is a powerful stochastic search method, which has been applied to a wide range of different areas, including mathematics, physics and mathematical programming [6]. It is used in this paper in an attempt to address an NP-complete combinatorial problem, which is another area where the method has been successful in the past. [6]

When applied to a combinatorial optimization problem, simulated annealing can be seen as a method for exploring a solution space S. The idea is simple, start at a random solution to the problem and make a small random change to the solution, if the new solution is better than the previous one, then replace it with the new one. If it is worse, then replace it with some probability dependent on how much worse it is and on how many iterations have passed.

In order to apply simulated annealing to a combinatorial problem, a utility function, a neighbourhood function and a cooling schedule need to be specified. The utility function assigns a real number to any given solution as a measure of how good the solution is. The neighbourhood function describes the neighbourhood of a solution and is used to generate new random solutions which are spatially close to the given solution in S. Finally the cooling schedule describes how the probability of accepting a worse solution is lowered over time.

In order to get good performance out simulated annealing, all of the above functions need to be adapted to the problem. The cooling schedule in particular, needs to be carefully adapted to the problem and occasionally even to particular instances of the problem [6]. When used in practice, a self-adapting schedule is often employed to help alleviate this problem.

2.4 Evaluation

In order to evaluate any of the proposed algorithms on actual data, a reliable way of measuring the performance of Java programs was required. However, due to the dynamic nature of program execution and compilation in Java and the Java-virtual-machine (JVM), it is often far from easy [7]. Qualitative benchmarking requires careful consideration of the sources of non-determinism, for instance adaptive compilation and memory management [8]. The code segment to be measured (payload) and the surrounding framework need to be properly designed to mitigate these problems.

2.4.1 Just-In-Time Compilation

In contrast with languages like C which is only compiled before runtime[8], Java programs are compiled in stages, using the Just-in-time (JIT) compiler to further optimize the program during execution [7]. Java provides several different options for JIT optimization to either delay or force early optimization. To avoid further compilation during a measurement, a warm up phase is often employed to allow Java to complete all compilation stages.

2.4.2 Memory Management

The memory management system in Java uses automatic garbage collection (GC) to manage its heap, i.e. the memory area that JVM uses to store data. During program execution, the GC is invoked repeatedly to clean up unused memory, causing a small performance loss each time. If the GC is heavily invoked during a measurement the results may become less consistent. An important choice is therefore whether to include GC performance in the evaluation or not [7]. Not including GC performance means that GC has to be forcibly invoked in between runs of the benchmarking payload. Then since the heap is cleaned before each execution the intention is that JVM will not invoke GC during any payload execution. However, after some initial analysis it was decided to include GC in the benchmarks mainly based on two reasons. Firstly, the analysis showed that even with thousands of payload executions per second, none of the algorithms were particularly memory allocation intensive. Thus the need for GC was low and did not affect the measured performance significantly. Secondly, in a real environment, the high volume of executions could be similar to the load of a dedicated server, where GC performance of course would be relevant, even if only used sparingly.

2.4.3 Programming Pitfalls

One of the more common issues with the payload design is called dead code elimination (DCE) [9]. DCE is an optimization method in the Java compiler which tries to identify and skip code that does not affect the output of a program. When a payload is created it is often extracted from a larger segment where other parts which depends on the output of the chosen payload are excluded. In extreme cases this means that the Java compiler may decide to skip most of the payload since its output is not used. The main way to prevent DCE is therefore to include some code that consumes the payload output. To implement the consumption of payload output, a benchmarking framework was used, see section 2.4.4.

Another potential pitfall is constant folding (CF), this too, a Java compiler optimization technique [9]. The purpose of CF is to avoid unnecessary computations that can be replaced by constants. Sometimes CF may be desired to reflect real-word performance but if the payload only depends on constants, then CF may cause the payload to be evaluated only once. However, undesirable CF was not an issue for the evaluation since all payloads depended on external data.

2.4.4 Java Microbenchmarking Harness

The Java Microbenchmarking Harness (JMH) is a benchmarking tool developed by Oracle, released in 2013. It aims to address the many issues with reliable benchmarking in Java. For instance, JMH contains a very simple way of consuming payload output to avoid DCE and implement warm up phases, reducing the need for additional error prone framework. Furthermore, JMH provides some statistical analysis in the form of confidence intervals and standard deviation. We therefore used JMH for all our performance benchmarks in order to ensure accurate and consistent results.

3

Linear Search

The obvious approach to solving the weighted set containment problem is to simply do a linear search through all the candidate sets and verify whether each element exists in the query set. We refer to the operation of verifying whether a given element exists in the query set as a *containment operation*. The number of *containment operations* performed is then, in the worst case where there are no disqualifications, $\sum_{i=1}^{n} m_i$.

If we have no prior information about the data, we cannot do better than $O(nm)$ *containment operations*. This is because every candidate could potentially be part of the solution and any element in a candidate set could potentially disqualify it. However, the speed of a *containment operation* significantly depends on the data structure used for the query set.

3.1 Baseline

In Volvo's current algorithm, which we refer to as *Baseline*, the elements are stored in a list. As a result, *containment operations* on the query set are performed by linearly comparing every element in the query set to the input element until an identical element is found. This results in at most $|Q|$ comparisons. The total number of comparisons performed is therefore $\sum_{i=1}^{n} m_i|Q|$ in the worst case. Hence *Baseline* has a time complexity of $O(|Q|nm)$ comparisons.

3.2 Hashed Linear

We try an improved version of the linear algorithm which stores the query set in a hash table. If the elements of the query set are stored in a hash table, then we can perform a *containment operation* in $O(1)$ time on average, assuming we can perform the arithmetic operations required for hashing in $O(1)$ time for arbitrarily large numbers. Therefore when a linear search is done using a hash table, we achieve an average time complexity of $O(nm)$, which is as mentioned previously, the optimal time complexity if no preprocessing has been done.

3.3 Lazy Fetch

In a situation where the data set is not yet cached, the time it takes to perform an $O(nm)$ search is almost negligible in comparison to the time it takes to fetch the data. For this reason, we try an algorithm which aims to minimize the amount of data fetched. The idea is to fetch only a few elements at a time and if a candidate gets disqualified then the rest of its elements are not fetched.

4

Set-trie

In order to improve the running time of the algorithm, we explore the possibility of preprocessing the data using a set-trie structure. A weighted set containment query can be seen as a traversal through a tree of candidate sets, where every node is an element and every path from the root node to an end node represents a candidate set. Linear search can be seen as the traversal of an uncompressed set-trie. We can use this to compare the two approaches, figure 4.1 shows an example of this.

By representing the data as a trie, redundancies can be exploited to compare elements that occur in several different candidate sets at the same time. This results in fewer comparisons and allows many candidate sets to be disqualified simultaneously. Therefore, if the data has a lot of redundancy, we should see a significant increase in the performance of the algorithm. Figure 4.1 shows an example of a trie, notice that several candidate sets share the same nodes. However, notice also that not all redundancies are removed.

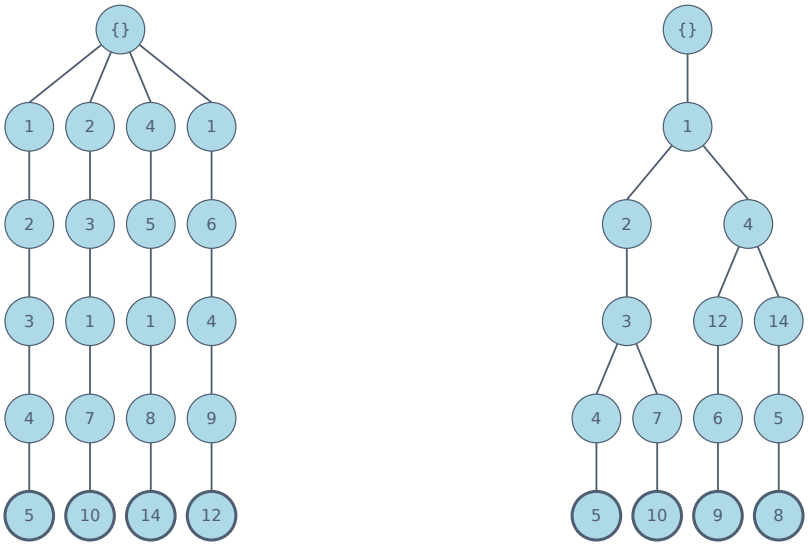


Figure 4.1: On the left, a graph representation of a linear search with the set family $\{1, 2, 3, 4, 5\}$, $\{1, 2, 3, 7, 10\}$, $\{1, 4, 5, 8, 14\}$, $\{1, 4, 6, 9, 12\}$. On the right, a set-trie containing the same elements. Every node is an element and every path from the root node to an end node represents a candidate set. The end nodes are indicated by a thicker outline.

Since a set-trie compresses the data, this approach should also result in the trie taking up less space in memory compared to loading the entire family of candidate sets. This is of particular interest for very large data sets where not everything can be cached in RAM.

4.1 Search

We try two different search strategies. In both of them, the query set is stored in a hash table. This section presents the two proposed traversal strategies in detail.

4.1.1 Simple Traversal

We can solve the weighted set containment problem by recursively exploring the trie. For any explored node, if it does not exist in the query set, then disqualify it and do not explore any of its children. If the node is an end node and it is not disqualified, then the path between it and the root node represents a potential solution and therefore we calculate its score. If the new score is higher than the currently highest, then replace the so far best path with the new path. Continue until there are no more nodes to explore.

Whether it is explored in breadth-first or depth-first order is irrelevant since the same amount of nodes will still be compared. Algorithm 1 shows pseudo code for exploring the trie recursively. Notice that all children of all elements that are contained in the query set need to be traversed. In a regular trie retrieval, this is not necessary, since we simply choose the child with the first character in the query string. However, since the search is not for prefixes, but for subsets, all children need to be checked.

This puts the simple traversal at a worst case running time complexity of $O(|T|)$, where $|T|$ is the number of nodes in the set-trie T. This worst case scenario is when all the candidate sets are subsets of the query set and we therefore need to explore the entire trie.

Algorithm 1 Simple Traversal

```
1: function searchTrie(node, querySet)
2:     if not querySet.contains(node) then
3:         return {[], −infinity}
4:     else if node.isLeaf then
5:         return {[node], node.weight}
6:     else
7:         bestNodes ←[]
8:         bestScore ← −infinity
9:         if node.isEndNode then
10:            bestNodes ←[node]
11:            bestScore ←0
12:        for all child in node.children do
13:            {newNodes, newScore} ← searchTrie(child, querySet)
14:            if newScore > bestScore then
15:                bestNodes ← newNodes
16:                bestScore ← newScore
17:            else if newScore = bestScore then
18:                bestNodes.addAll(newNodes)
19:        return {bestNodes, bestScore + node.weight}
```

4.1.2 Score Bounded Search

To improve the search speed we propose a branch-and-bound search strategy. Branch-and-bound is a search strategy used to explore a solution space, where solutions that cannot be optimal are pruned away [10]. The idea is to avoid traversing branches that cannot possibly contain any candidate set with a higher score than the best one found so far. This optimization however, comes at the cost of memory.

Given a node x, let $S_{max}(x)$ be the maximum potential score for any path from x to some end node.

$$S_{max}(x) = w(x) + max(\{S_{max}(c) : c \in x.children\}) \tag{4.1}$$

Since $S_{max}(x)$ is independent of Q, it can be precomputed. We take advantage of this fact by letting every node x store the value of $S_{max}(x)$. Additionally, each node x stores its children $c \in x.children$ in descending order from highest to lowest value of $S_{max}(c)$. An example of such a set-trie can be seen in figure 4.2. It would suffice to save the maximum potential score of every branch, not every node, this is overlooked here.

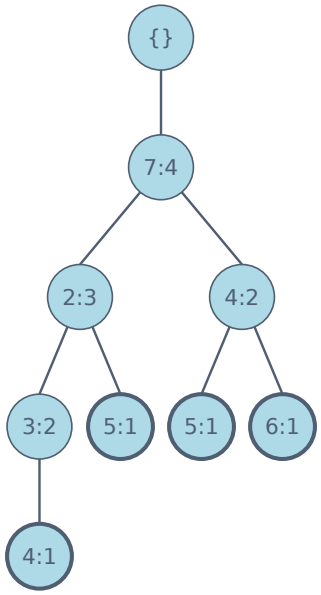


Figure 4.2: A score-bounded set-trie, containing the candidate sets $\{7, 2, 3, 4\}$, $\{7, 2, 5\}$, $\{7, 4, 5\}$, $\{7, 4, 6\}$. The maximum potential score is written to the right of every element number. The weights are not visualized, but are all set to one.

For any node x and child $c \in x.children$, let c' be the next child of x according to the defined order. Since the order is descending according to maximum potential score, if we find a valid path from c to any end node with the score s and $s > S_{max}(c')$ there is no need to explore c' or any subsequent children of x. To utilize this, the search has to run in depth-first order, so that a high scoring solution can be found early on. As an example, a search through figure 4.2 could visit the path $(7, 4), (2, 3), (3, 2), (4, 1)$ and would not consider any other nodes since the nodes in the chosen path have the highest maximum potential score. Pseudocode for this approach is presented in algorithm 2.

Algorithm 2 Score Bounded Traversal

```
1: function searchSBTrie(node, querySet)
2:     if not querySet.contains(node) then
3:         return {[], −infinity}
4:     else if node.isLeaf then
5:         return {[node], node.weight}
6:     else
7:         bestNodes ←[]
8:         bestScore ← −infinity
9:         if node.isEndNode then
10:            bestNodes ←[node]
11:            bestScore ←0
12:        for all child in node.children do
13:            if child.Smax < bestScore then
14:                break
15:            {newNodes, newScore} ← searchSBTrie(child, querySet)
16:            if newScore > bestScore then
17:                bestNodes ← newNodes
18:                bestScore ← newScore
19:            else if newScore = bestScore then
20:                bestNodes.addAll(newNodes)
21:        return {bestNodes, bestScore + node.weight}
```

4.2 Optimal Set-trie Problem

The most important problem to solve for any set-trie based approach is how to compute the optimal set-trie in terms of search time, from a family of candidate sets. Too simplify the problem, we estimate the search time by the number of traversed nodes in a search.

Since the number of traversed nodes largely depends on the query set Q used in the search, we can at best compute the expected number of traversed nodes by randomly sampling query sets from the data set. In order to do this, we define a binary random variable $D_{x_i} = \{0, 1\}$ for each unique element $x_i \in U$, where 0 represents $x_i \in Q$ and 1 represents $x_i \in Q$. The probability that an element x will be disqualified given a randomly sampled query set is then $P(D_x = 1)$, which we refer to as the disqualification probability of x. While we do have access to a large data set of query sets, these sets vary and new ones are created constantly, hence the accuracy of these estimates is questionable.

In any search, at least the first node $x = T.root$ is checked, there is then a $P(D_x = 0)$ probability that the search continues, in which case all of x's children need be to checked. Initially we consider the case where the random variables are independent.

Let $U_i(x)$ be the expected number traversed nodes for a search starting at the node x. We can define this function recursively:

$$U_i(x) = 1 + P(D_x = 0) \sum_{c \in x.children} U_i(c) \tag{4.2}$$

Thus, if the random variables are independent, then the optimal set-trie is the trie that minimizes $U_i(T.root)$. Note that higher disqualification probabilities closer to the root of a set-trie will result in fewer expected number of traversed nodes. However, if the random variables are not independent we can modify function 4.2 to handle conditional probabilities. Since traversing a node x means that all nodes between x and the root node were not disqualified, we instead use the probability of disqualifying x given the event that all nodes between the root node and x were not disqualified. Let $U_d(x, X)$ be the expected number traversed nodes for a search starting at the node x, where X is a conditional event. We define $U_d(x, X)$ similar to function 4.2:

$$U_d(x, X) = 1 + P(D_x = 0 \mid X) \sum_{c \in x.children} U_d(c, (D_x = 0) \cap X) \tag{4.3}$$

Similar to function 4.2, the optimal set-trie is the trie that minimizes $U_d(T.root, \Omega)$, where Ω is the sample space. We use Ω as the initial conditional event since we cannot assume anything prior to the search.

However, it would be safer to assume that we know nothing about the query set. Then we can not know which elements are more likely to lead to a disqualification, hence we do not need to consider the order in which the elements are checked. In this case, the optimal set-trie is the trie with the fewest nodes possible, we call the problem of finding such a trie the minimum set-trie problem. We show that this problem is NP-complete, see section 4.2.1. Furthermore, since minimizing $U(T.root)$ can be seen as a more general case of the same problem, consider the case when $\forall x \in T, P(D_x = 1) = 0$, then the two problems are equivalent. Therefore minimizing $U(T.root)$ is at least as hard.

Additionally, the search space of all possible set tries is vast. All possible set-tries that represent a given family of candidate sets can be seen as a combination of internal candidate set permutations. By inserting all candidate sets into an empty trie, according to their internal order, the corresponding trie is constructed. The number of unique orderings of the candidate sets is $\prod_{i=1}^{n} m_i!$. The worst case time complexity of exhaustive search is therefore $O(m!^n nm)$, where building a single set-trie according to some permutation is done in $O(nm)$.

Since the problem is NP-complete and since the search space is large, we attempt to approximate the optimal solution either by some type of greedy algorithm or with a metaheuristic.

4.2.1 Proof of NP-completeness

Definition 4.2.1 (The minimum set-trie problem)**.** *Given a universe $U = \{x_1, x_2, ..., x_\mu\}$, a family of subsets $F = \{C_1, C_2, ..., C_n\}$ where $C_i \in F$, $C_i \subseteq U$ and an integer k, is it possible to construct a set-trie T of size k?*

Definition 4.2.2 (The vertex cover problem)**.** *Given a graph $G = (V, E)$ and an integer k, is it possible to construct a set S of size k such that $\forall(u, v) \in E, u \in S \vee v \in S$?*

Theorem 4.2.1. *The minimum set-trie problem is NP-complete.*

Proof. The problem is in NP since, for any instance F and solution T, a verifier can check in polynomial time that T is of size k and that each $C_i \in F$ is represented in T.

We continue by showing that vertex cover $\leq_p$ minimum set-trie. Given an instance of vertex cover, i.e. a graph $G = (V, E)$ where $m = |E|$ and an integer k, we construct an instance of the minimum set-trie problem in polynomial time. Let $U = V$, for every edge $e_i = (u, v) \in E$ we create a set $C_i = u, v \subseteq U$ resulting in a subset family $F = \{C_1, C_2, ..., C_m\}$ of size m. **Claim:** there exists a vertex cover of size k if and only if the constructed set-trie is of size $k + m + 1$. To show this, we prove the equivalence in both directions.

vertex cover ⇒ minimum set-trie:
Given an vertex cover ($G = (V, E)$) with a solution S of size k, we have a corresponding instance of set-trie with the subset family F. From the definition of vertex cover we have that every set $C_i \in F$ contains at least one element in the vertex cover. Therefore, we can construct a set-trie such that the first layer consists only of the vertices in the vertex cover, thus there are k nodes in the first layer. Since all sets in F are unique, the second element will always create a new node in the second layer, thus there are exactly m nodes in the second layer. Hence the total number of nodes in the trie is $k + m + 1$ including the root node.

minimum set-trie ⇒ vertex cover:
Given a set-trie (F) with solution T of size $k + m + 1$, we have a corresponding instance of vertex cover with $G = (V, E)$. Since all sets in F are unique and the set-trie represents all m of them, there must be exactly m nodes in the second layer. Furthermore there is exactly one root node, therefore there is $k+m+1-(m+1) = k$ nodes in the first layer. Every set $C_i \in F$ contains at least one node in the first layer, therefore every edge $e_i \in E$ is connected to at least one node with an element in the first layer. Thus, the nodes in G that contain the elements in the first layer is a vertex cover of G. Hence G has a vertex cover of size k.

Since our earlier claim holds, the construction algorithm runs in polynomial time and minimum set-trie is in NP, we conclude that the minimum set-trie problem is NP-complete. □

4.2.2 Related problems

Although we have not been able to find any research on the minimum set-trie problem, there are some related areas which deal with somewhat similar problems. One such problem is finding the optimal decision tree, which is also an NP-hard problem for most criteria [11]. There are however numerous differences between these two problems. One important difference being that decision trees are usually used for classification and are typically not optimized for size of the tree. Viewing the weighted set containment problem as a classification problem is problematic. There are for example situations where multiple candidates should be returned, in which case we would need to classify the query set as multiple candidates.

4.3 Greedy Approach

We attempt to find the minimum set-tries using a greedy algorithm. The algorithm uses the heuristic of always inserting the most common element among the remaining candidate sets. We first find the most common element e, we then insert e into the trie and split F into two parts, the candidate sets which contain e and those that do not. We then remove e from all candidate sets. If any candidate set has no more elements left, we mark the inserted node e as an end node. This algorithm is then recursively called on the two new sets families, see algorithm 3 for simplified pseudocode which illustrates the principle. For the complete optimized version, see algorithm 4 in appendix A.

Algorithm 3 Greedy Set-trie Build

```
1:  procedure buildGreedySetTrie(parent, candidateSets)
2:      chosenElement ←most common element among candidateSets
3:      newNode ← parent.addChild(chosenElement)
4:      chosenCandidateSets ←[]
5:      remainingCandidateSets ←[]
6:      for all set in candidateSets do
7:          if set.contains(chosenElement) then
8:              set.remove(chosenElement)
9:              if set.isEmpty then
10:                 newNode.markAsEndNode
11:             else
12:                 chosenCandidateSets.add(set)
13:         else
14:             remainingCandidateSets.add(set)
15:     buildGreedySetTrie(newNode, chosenCandidateSets)
16:     buildGreedySetTrie(parent, remainingCandidateSets)
```

The algorithm can be seen as repeatedly approximating the minimum hitting set for every layer. The minimum hitting set in this context is the smallest set of elements

such that every candidate set contains at least one element in the set. The problem of finding the minimum hitting set is NP-complete but can be approximated by a factor of the harmonic number $H(n)$ by repeatedly selecting the most frequently occurring element until a hitting set is found [12]. Note that it is precisely this rule that our greedy algorithm uses.

After computing a hitting set, the candidate are divided according to which element in the hitting set they contain. Candidate sets that contain more than one element in the hitting set, are chosen to be put in the branch with the most frequent element. This decision is rather arbitrary and we propose a few alternatives, see section 4.4.

4.3.1 Implementation

The algorithm can be efficiently implemented using a hash table based data structure for sets. By doing so we achieve $O(1)$ time on average for the add, get, remove and contain operations, assuming we can perform the arithmetic operations required for hashing in $O(1)$ time for arbitrarily large numbers. If the size of the hash tables are kept small enough or if they are supplemented with a linked list, they can be iterated through in linear time with respect to the number of elements which they represent.

In the case where there exists a set S such that $|S| > 1$ and S is a subset of every candidate set, it is unnecessary to recursively call the algorithm for each element in S since we already know that all of these elements will be inserted in a sequence. Therefore, the algorithm can be made more efficient if it checks for such sequences and when they are found, it simply inserts the whole sequence in the trie. See algorithm 4 in appendix A for this optimized version.

4.3.2 Time Complexity

To analyze the time complexity of the optimized version of the algorithm described in section 4.3.1, let all candidate sets be of equal cardinality.

If there are elements that occur in every candidate set, then they are all inserted in $O(nm)$ time. Otherwise the most frequently occurring element is found and inserted, which also takes $O(nm)$ time. Therefore each recursive call takes $O(nm)$ time.

Consider a path P where every node has exactly one child. We know that any such path where $|P| > 1$ is inserted by a single recursive call. Then let T' be the trie where any such a path is compressed into a single node. Since there are no paths P in T' such that $|P| > 1$, every other node in any branch has at least two children

Since there are no paths P in T' such that $|P| > 1$, every other node in any branch has at least two children. Therefore the number of nodes in layer j is at least twice the number of nodes in layer $j - 2$. As a result, the total number of nodes in T' is at most $2 * (n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + ... + 1) = 4n$. Since each node in T' corresponds to a single recursive call and every such call takes $O(nm)$ time, the worst case time complexity is $O(|T'|nm) = O(n^2m)$.

4.3.3 Alternative Implementation

We can implement the algorithm in a theoretically more efficient way, assuming $m < n$, although it comes at the cost of more overhead. In this implementation, we maintain an index of the most frequently occurring elements, that way the index does not need to be recomputed for every recursive call. Specifically, an inverted index of the set family allows for finding the frequency of an element and what sets contain it in $O(1)$ time. In addition, an index mapping the number of occurrences to their corresponding elements e.g map the number five to all elements that occur in exactly five candidate sets, allows for finding the most frequent element in $O(1)$ time.

For any inserted element, we need to update the index for the rest of the layer and create a new index for the subtree under the inserted element. Both of which take linear time with respect to the number of elements represented by the subtree. Since layer i needs to represent every candidate set and i elements have been removed from each candidate set, $n * (m - i)$ elements are represented by its subtrees. It therefore takes $O(nm)$ time to insert all nodes in a layer. Since all candidate sets are of length m, there are m layers in the trie. Thus inserting all the nodes in the trie takes $O(nm^2)$ time, which is as stated earlier, preferable to $O(n^2m)$ time of the original greedy if $m < n$.

4.3.4 Optimality

While the algorithm does sometimes generate an optimal solution, it is not guaranteed, which is easily shown by the following counterexample. Consider the set family F consisting of four sets:

$$A_1 = \{1\} \qquad A_2 = A_1 \cup \{2\}$$
$$A_3 = \{3, 4\} \qquad A_4 = A_1 \cup A_3$$

Using the greedy algorithm, the most common element 1 is selected first. The only set that does not contain 1 is A_3 which means that the set-trie will have two branches, one representing A_3 and one representing A_1,A_2 and A_4. The total size of the greedy solution is therefore $|T_G| = |A_3| + |A_4| + 1 = 6$. However, if we instead split the root into two branches, one for A_1 and A_2; and one for A_3 and A_4, then we end up with the smaller size $|T_{OPT*}| = |A_4| + |A_2| = 5$.

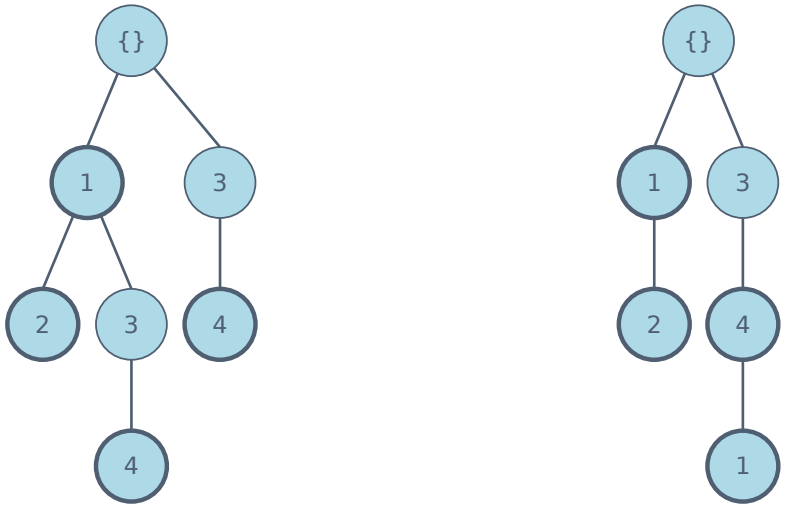


Figure 4.3: A simple example of how the greedy algorithm can create a suboptimal set-trie. On the left, the set-trie constructed by the greedy algorithm and on the right the minimum set-trie.

Not only is the greedy algorithm not optimal, in fact it does not approximate the optimal solution by any constant factor.

Theorem 4.3.1. *There cannot exist a constant factor c, such that the greedy algorithm is a c-approximation.*

Proof. We extend the previous example with an arbitrary number of sets. We begin by defining the following sets given two positive integers k and t:

$$\begin{aligned} K^* &= \{x \in \mathbb{N} : x < k\} \\ K &= K^* \cup \{k\} \\ S &= \{x_1, x_2, ..., x_t\}, \text{where } K \cap S = \emptyset \\ A_i &= \{x \in \mathbb{N} : x \leq i\} \cup S \end{aligned} \tag{4.4}$$

The set-trie instance then consists of the candidate sets $F = \{K^*, K\} \cup \{A_i : 0 \leq i < k\}$. Again, using the greedy algorithm we see that the only set which does not contain the most common element 1, is A_0. Therefore we split F into two branches, one for A_0 and one for $\{K^*, K\}\{A_i : 1 \leq i < k\}$. We then recursively repeat the process for every element i and set A_i where $1 \leq i < k$ resulting in the total size $|T_G| = kt + k = k(t + 1)$. For the better solution T_{OPT*}, we split the root into two branches, one for $\{K^*, K\}$ and one for $\{A_i : 0 \leq i < k\}$ resulting in the total size $|T_{OPT*}| = k + t + (k - 1) = 2k + t - 1$.

Example

Consider an instance of the sets in equation 4.4 where $S = \{4, 5\}$ and $k = 3$:

$$\begin{aligned} K^* &= \{1, 2\} & K &= \{1, 2, 3\} \\ A_0 &= \{4, 5\} & A_1 &= \{1, 4, 5\} \\ A_2 &= \{1, 2, 4, 5\} \end{aligned}$$

The resulting set-tries, shown in figure 4.4, highlight the non-optimal construction pattern of the greedy algorithm which repeats the elements of S exactly k times.

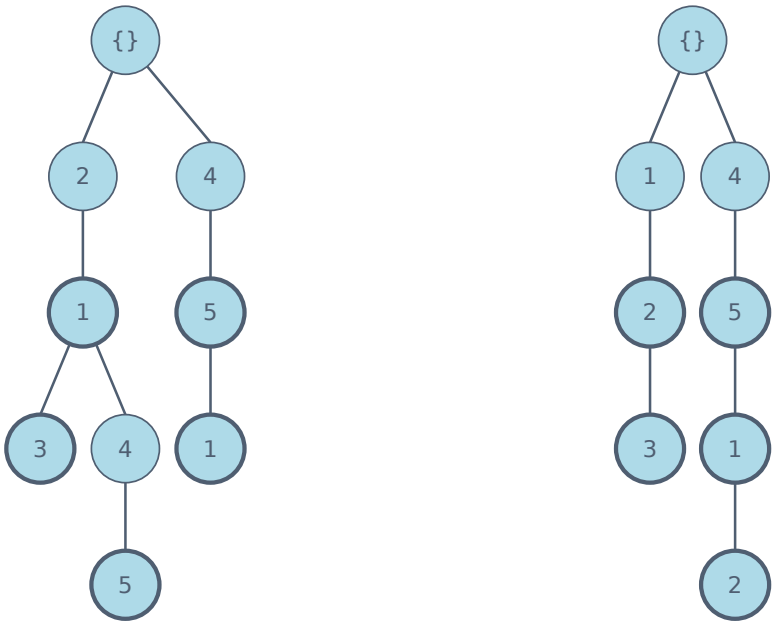


Figure 4.4: An example of how the greedy algorithm can create a suboptimal set-trie. On the left, the set-trie constructed by the greedy algorithm and on the right the minimum set-trie.

Let T_{OPT} be the actual optimal solution such that $|T_{OPT}| \leq |T_{OPT*}|$. Then, let t approach infinity in the ratio between $|T_G|$ and $|T_{OPT*}|$ which gives us:

$$\lim_{t\to\infty} \frac{|T_G|}{|T_{OPT*}|} = \lim_{t\to\infty} \frac{k(t+1)}{2k+t-1} = k \Rightarrow \lim_{t\to\infty} \frac{|T_G|}{|T_{OPT}|} \leq k$$

Since k can be chosen to be arbitrarily large, there cannot exist a constant factor c such that the greedy algorithm is a c-approximation. □

Although the greedy algorithm does not guarantee a constant factor approximation, we can at least be sure of an m-approximation or an n-approximation, see theorem 4.3.2.

Theorem 4.3.2. *Any valid set-trie for a set family where all candidate sets have equal cardinality is an m-approximation and an n-approximation.*

Proof. A valid set-trie cannot contain more than $nm + 1$ nodes, let T_{MAX} denote such a set-trie. The optimal set-tie denoted T_{OPT} cannot have fewer than $n + m$ nodes, which gives us the approximation factor:

$$\frac{|T_{MAX}|}{|T_{OPT}|} = \frac{nm + 1}{n + m}$$

Since $n > 0$ we have that $n + m \leq m + 1$ and $nm + 1 \leq nm + n$, thus:

$$\frac{|T_{MAX}|}{|T_{OPT}|} = \frac{nm + 1}{n + m} \leq \frac{nm + n}{m + 1} = n$$

We can show that $\frac{|T_{MAX}|}{|T_{OPT}|} \leq m$ using the same argument. □

4.4 Greedy Variations

To address the fact that the greedy algorithm can construct badly performing tries under certain circumstances, a few variations designed to deal with these special cases were tried. In particular, the counterexamples presented in section 4.3.4 seem to indicate that bad approximations appear when long sequences are represented in more than a single subtree. This happens because the greedy algorithm divides the candidate sets up in a naive way. Specifically, all candidates that contain the most frequently occurring element are all placed under the same subtree, even if a candidate shares a lot more elements with some other subtree.

4.4.1 Group Based Approach

The first greedy variation, called group-greedy, is based on the observation that it is frequently the case in the data set that certain elements occur exclusively in groups. If we find these groups, we can prevent the greedy algorithm from splitting them up and thus avoid long sequences being represented in several subtrees. In order to incorporate this idea we replace the previous heuristic with a simple function that allows us to compare the outcome of inserting groups of variable length. The function describes the number of elements that are combined given a group X and F', the set of candidate sets that contain X:

$$G(F', X) = |X|(|F'| - 1)$$

Note that if $|X| = 1$ then the group variation is exactly the same as the original greedy algorithm. The groups that are compared are all groups of elements that exclusively occur together. All such groups can be found in $O(nm)$ time, using an inverted index and by combining identical values.

4.4.2 Intersection Based Approach

We also try an intersection based approach, which attempts to alleviate the greedy algorithms shortcomings using an intersection based heuristic. Just like in the greedy algorithm, we recursively select the most frequently occurring element until all the candidate sets contain at least one of selected elements, we refer to this as computing a hitting set. However, after a hitting set is computed, the candidate sets are partitioned in a different way. Firstly, all candidate sets that contain exactly one of the elements in the hitting set are placed under that element. Any candidate set that contains more than one element in the hitting set is placed under the element whose corresponding group of candidate sets score the highest according to an intersection based heuristic.

We propose the following two heuristics $H_{max}(C, F')$ and $H_{avg}(C, F')$, where C is the candidate set to be placed and F' is a potential group of candidate sets:

$$H_{max}(C, F') = max(\{|C \cap C'| : C' \in F'\})$$
$$H_{avg}(C, F') = \frac{1}{|F'|} \sum_{C' \in F'} |C \cap C'|$$

The functions are similar but H_{max} computes maximum intersection size, while H_{avg} computes the average intersection size.

This variation of the greedy algorithm requires a lot of additional computations but should help in situations where very long sequences are represented in more than a single subtree. Although it should be noted, that this solution by no means guarantees to always solve the problem optimally, it merely serves as a heuristic guess.

4.4.3 Disqualification Probability Based Approach

We try three algorithms that are based on the probabilistic modelling introduced in section 4.2. These methods are based on the assumption of independent events, and hence are based on the objective function U_i (see function 4.2). There are a number of reasons we make this assumption. Firstly, since we do not have access to a large amount query sets we are unlikely to be able to approximate the complete probabilistic model with any reasonable accuracy. Secondly, we measured a 0.98 correlation between U_i and U_d (see function 4.3) on tries generated by the greedy algorithm and hence we have reason to believe that the assumption won't affect the result to any large degree, see figure 4.5.

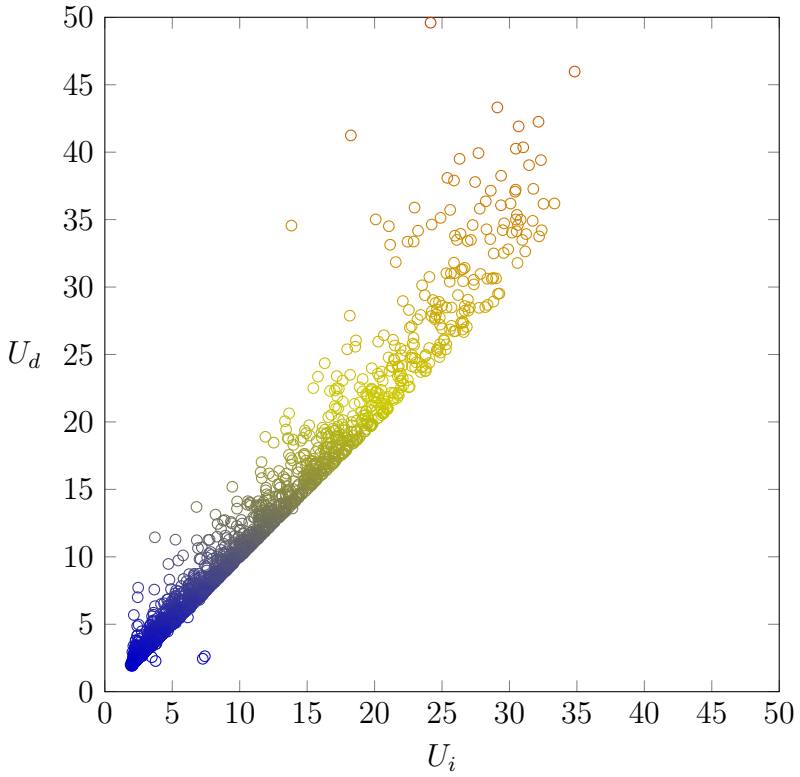


Figure 4.5: The relation between the objective functions U_i and U_d (see functions 4.2 and 4.3). The data points are represents set-tries for approximately 30,000 set families built by the greedy algorithm, using a sample of 2000 query sets for the estimation of disqualification probabilities. The color highlights the value of U_d from blue (low) to red (high).

To minimize the objective function 4.2 introduced in section 4.2, we try two additional variations that both use disqualification probability to decide which element to insert. One of the variations simply inserts the element with the highest disqualification probability, with no consideration of frequency. The other only uses disqualification probability as a tiebreaker, picking the element with the highest probability if there are multiple elements with the same frequency.

4.5 Global Ordering

Contrary to the greedy algorithm we propose in this thesis, many authors of similar trie based data structures, use some type of global ordering to build a trie from a family of sets. The idea is that by ordering the candidate sets according to a global order of all unique elements, prefixes of elements will be combined and the resulting trie will have fewer nodes. While some simply assign an arbitrary order [2], other authors [5] and [1], use a global order based on the frequency of elements for their set containment algorithms.

We try a global order in both ascending and descending order of frequency, since these were the approaches used by Jampani and Bouros in their state of the art set containment join algorithms. We further expand on global ordering by introducing an order based on disqualification probability.

An advantage of global ordering based approaches is that the tries can be constructed relatively quickly. Specifically, a trie based on a global ordering can be constructed in $O(lg(m)nm)$ time by sorting every candidate set.

4.6 Simulated Annealing

Since we cannot find the optimal solution in any reasonable time frame and since guaranteeing a reasonable approximation seems very difficult, we attempt a heuristic approach using the optimization method simulated annealing, see section 2.3 for an explanation of the concept.

In this thesis, simulated annealing is used for two different approaches, one which explores the space of all solutions and one which explores the space of all global orders of unique elements. The two approaches uses the same cooling schedule and utility function but use slightly different representations and neighbourhood functions.

The utility function is simply the negated number of nodes in the constructed trie, which can be computed by constructing the trie. Furthermore both a linear and an exponential cooling schedule were tried with experimentally determined parameters.

4.6.1 All Possible Solutions

In order to search the space of all possible tries, a solution can be described as a combination of internal candidate set permutations. The corresponding trie can then be constructed by inserting all the elements in every candidate set according to their internal order. This definition means that we are exploring a solution space of size $\prod_{i=1}^{n} m_i!$. In order to explore this vast solution space, we generate a neighbouring solution by randomly selecting two elements from a randomly selected candidate set and then reverse the order of the elements between them. This reversal of the path between any two elements is a commonly used neighbourhood function which has been used successfully in other combinatorial NP-complete problems such as the travelling salesman problem [13].

4.6.2 All Solutions Based on Global Orderings

If we assume that there are always at least some good solutions which are based on a global ordering, it could then prove useful to explore all solutions which are global orderings, due to the greatly reduced size in solution space.

To explore the set of all tries that can be constructed by a global ordering, a solution is represented as a permutation of all unique elements. To construct the corresponding trie, all candidate sets are sorted according to the permutation and are then inserted according to their internal order. This representation corresponds to a solution space of $|U|!$, which is significantly smaller than the other approach, especially for redundant data sets where the number of unique elements is low. This smaller solution space should make the optimization method converge much faster.

To generate a neighbouring solution, two random elements in the permutation of all unique elements are chosen and the order of the elements between them is reversed.

To help the algorithm converge, a form of short term memory was used. This was implemented by having the method revert back to the best solution found over the last 100 iterations every 100^{th} iteration.

5

Results

The following chapter presents the results and analysis of the proposed solutions to the weighted set containment problem. During the initial evaluation of the baseline algorithm it was discovered that in practice, the data fetch time, that is, the time it took to retrieve the data necessary to perform a search was very large. In the case of the test suite, the data fetch time was about 1000 times larger than the search time for the baseline algorithm. A brief attempt was made to reduce this difference by implementing variations of lazy data fetch in section 3.3. However, since the amount of data needed to store all set families could be stored in memory, the data fetch time was removed from the performance evaluation. The remaining linear search approaches and all set-trie variations are therefore evaluated only on the search time with cached data.

The targeted data set consists of about $100,000$ set families and several million different query sets. While the size of the query sets is relatively consistent at about $300 - 900$ elements, the size of the set families range from 1 candidate set to more than $10,000$. The candidate sets consistently contain roughly $10 - 20$ elements. We evaluated our methods with a representative subset sampled from this data set with 100 set families and 10 query sets.

5.1 Linear Search

In this section, the performance of the linear search based approaches are evaluated. Firstly the search time of the two first approaches are evaluated, under the assumption that the data is cached in advance. Then, the search time of all the linear approaches are evaluated in the scenario where the data is not cached in advance.

5.1.1 Search Time on Cached Data

Table 5.1 shows the search time of our different linear search based approaches. *FastLinear* refers to an improved version of the baseline algorithm, which was implemented with more efficient data types.

All suggested methods, except *Baseline*, were based on an improved version of the data set, where some data types had been exchanged for more efficient alternatives.

FastLinear-Hash further builds upon the improvements introduced by *FastLinear* and adds hash tables to efficiently store the query sets. The change from linear lookup time in *FastLinear* to constant lookup time reduces the search time by a factor of 10.

Algorithm	Search Time	Ratio to Baseline
Baseline	149 386 µs	1×
FastLinear	47 741 µs	3×
FastLinear-Hash	5742 µs	26×

Table 5.1: The search time for linear search algorithms with cached data.

5.1.2 Search Time on Non-cached Data

In an attempt to reduce the significance of the data fetch time, two additional linear search algorithms were implemented, both employing a lazy data-retrieval strategy. Table 5.2 shows that the lazy variation perform significantly better on non-cached data compared to the standard linear algorithms.

Algorithm	Search Time	Ratio to Baseline
Baseline	64 521 ms	1×
FastLinear	64 357 ms	1×
FastLinear-Hash	64 302 ms	1×
LazyFetch	608 ms	106×

Table 5.2: The total search time for linear search algorithms with non-cached data.

5.2 Set-trie

The different set-trie approaches were evaluated and compared based on pre-cached search time, the total node count of the created tries and finally the time time it took to preprocessing the tries themselves.

5.2.1 Search Time

Using the simple traversal algorithm (see section 4.1.1), the search time for set-tries created by the greedy algorithm performed much better than linear search, up to 1800 times faster than the baseline algorithm, see figure below. This is undoubtedly in large part due to the redundancy in the data set. In fact, only 1.6% of all elements in the test suite were unique in any set family, which certainly explains the big increase in performance.

In addition to the set-trie compression, a few tricks that were specific to Volvo's particular data were used to to make the tries smaller. These tricks roughly halved

the search time of the tries. For instance there were certain elements that were present in every query set and had a weight of 0, which could be cut from the trie. However, even without these tricks, the greedy algorithm was still more than a 900 times faster than the baseline algorithm.

In table 5.3, we see that the attempted variations of *Greedy* ended up performing slightly better, except for the group based one. This indicates that the *Group-Greedy* algorithm is not finding the longer sequences which are found by the intersect based methods.

Algorithm	Search Time	Ratio to Baseline
Baseline	149 386 µs	1×
Group-Greedy	88 µs	1698×
Greedy	84 µs	1772×
Intersect-Avg-Greedy	81 µs	1837×
Intersect-Max-Greedy	80 µs	1863×

Table 5.3: The search time of the different variations of the greedy algorithm and how they compare to the search time of the original baseline algorithm previously in place at Volvo.

When the score-bounded search strategy was used (see section 4.1.2, the search time improved but only slightly, see table 5.4. Compared to the regular search strategy, we see an increase in performance of roughly $3-6\%$ for all algorithms except *Greedy*, which got slightly worse. The small size of this increase is not surprising, given that in the test suite only 0.76% of the candidate sets ended up with a unique potential score. Although uniqueness is not a perfect measure of how useful score-bounded search is, the extremely low percentage nevertheless hints at fairly uniform scores for different branches.

Something of note, is that the increase in performance for the *Intersect-Max-Greedy* algorithm was greater than for *Greedy*. This could be a coincidence of the particular data set, but it could also indicate that score-bounded search performs better with tries created with the *Intersect-Max-Greedy* algorithm.

Considering the additional memory and complexity needed to perform score-bounded search, it does not appear to be a practical approach, at least not for this data set. Although a data set with highly variable weights or lengths of the candidate sets, could benefit greatly.

Algorithm	Search Time	Ratio to Baseline
Baseline	149 386 µs	1×
Greedy - (score-bounded search)	85 µs	1751×
Group-Greedy - (score-bounded search)	79 µs	1902×
Intersect-Avg-Greedy - (score-bounded search)	76 µs	1955×
Intersect-Max-Greedy - (score-bounded search)	74 µs	2011×

Table 5.4: The search time of the different variations of the greedy algorithm using the score-bounded search strategy.

The approach of assigning a global order to the elements and then simply inserting the elements in that order was studied for comparison. This method appears to be the standard approach for creating prefix tries, it was employed by both Bouros et al. and Jampani et al., see section 2.2. In these papers, the elements are sorted according to their frequency in the set family, in ascending and descending frequency, respectively. Table 5.5 shows the search time for these strategies and the best performing method presented in this thesis for comparison.

The ascending frequency approach did not perform particularly well, probably mostly because it created relatively large tries. On the other hand, descending frequency performed well and is a simple algorithm with significantly shorter construction time. However, the *Greedy* algorithm still outperformed this approach by roughly 50%. We can conclude that the *Greedy* approach is the preferable one for a data set such as this one, at least when the preprocessing time of the trie is not critical.

Algorithm	Search Time	Ratio to Baseline
Baseline	149 386 µs	1×
GlobalOrder - (ascending frequency)	434 µs	344×
GlobalOrder - (descending frequency)	137 µs	1083×
Intersect-Max-Greedy - (score-bounded search)	74 µs	2011×

Table 5.5: The search time of the global order strategy, and the best performing approach presented for comparison.

The search time of the disqualification probability approaches that were tried are shown in table 5.6. The disqualification probabilities used here are overfit to this particular data set and would not be representative if used in practice. Despite this overfitting, the search time ends up significantly worse than when these probabilities are ignored. Even when the *Greedy* algorithm is used and the probabilities are only used as a tiebreaker, we see no improvement. This indicates that our approach of trying to minimize the set-tries was the correct approach. Although it should be noted that the *Greedy-DP* algorithm in table 5.6, did create larger set-tries than usual and still managed to get a decent search time average.

Algorithm	Search Time	Ratio to Baseline
Baseline	149 386 µs	1×
Greedy-DP	160 µs	935×
GlobalOrder-DP	128 µs	1167×
Greedy-DP-Tiebreaker	83 µs	1803×

Table 5.6: The search time of the different variations of disqualification probability. *Greedy-DP* refers to the approach of using the *Greedy* algorithm but using the disqualification probability instead of frequency to select elements. *GlobalOrder-DP*, sorts the elements with descending disqualification probability. The *Greedy-DP-Tiebreaker* algorithm is the same as the *Greedy* algorithm but in the case of a tie, the element with the highest disqualification probability is chosen.

5.2.2 Node Count

Table 5.7 shows the performance of the different algorithms, in terms of how large the tries became. The *Uncompressed tree* refers to a tree that does not utilize a trie's capability to compress redundant data, this tree serves as a reference point since traversing through it is equivalent to performing a linear search. Notice that the compression is significant, the best performing algorithm *Intersect-Max-Greedy* in terms of compression, reduces the number of nodes in the trie by a factor of 9.

The approaches based on simulated annealing ended up performing substantially worse than the *Greedy* approach, despite having a much slower preprocessing time. We could not get the approach which explored all possible set-tries to work at all, it never found any solution better than the initial order and only got worse over time. The version which explored all global orders, *SimulatedAnnealing-GlobalOrder*, worked much better and converged after roughly 20 000 iterations, although rarely at the optimal global order. These methods likely performed poorly due to the large search space of all possible set-tries. Furthermore, small changes in the representation could lead to much worse utility and small changes leading towards a better solution were not detectable at all by the utility function.

The descending version of *GlobalOrder* performed well, but still worse than any *Greedy* based approaches. The ascending version created very large trees and performed badly in terms of search time, which indicates that rarity in the set family did not correspond to a higher disqualification probability to any significant degree. Further evidence for this is the large size of *Greedy-DP*, which is almost as large as the ascending *GlobalOrder* algorithm, but still performs much better in terms of search time. This also means that the *Greedy-DP* did create large tries but still managed to get a decent search time. While *Intersect-Max-Greedy* performed the best on average, all *Greedy* based approaches were close enough to have singular instances where they performed the best.

Algorithm	Average Node Count
Uncompressed tree	2527.0
GlobalOrder - (ascending frequency)	1004.1
Greedy-DP	725.5
GlobalOrder-DP	578.9
SimulatedAnnealing-GlobalOrder	465.6
GlobalOrder - (descending frequency)	328.8
Greedy	293.2
Group-Greedy	288.4
Intersect-Avg-Greedy	282.8
Intersect-Max-Greedy	278.9

Table 5.7: The average number of nodes in the set-tries built by the different algorithms.

5.2.3 Preprocessing time

The time required to construct the tries is shown in table 5.8. Given that n is generally much larger than m in the test suite, we expected our index based alternative implementation of the *Greedy* algorithm to outperform the original implementation. However, this turns out not to be the case and is likely due to increased overhead. The intersection based variations on *Greedy* became much slower than the other *Greedy* variations, this is undoubtedly due to the numerous set intersection operations performed. Given that *Intersect-Max-Greedy* only performed slightly better than the other *Greedy* variations, regular *Greedy* or *Group-Greedy* are probably preferable in most scenarios. This is especially the case for very large set families, where the preprocessing time for *Intersect* variations might get too long to realistically compute.

The preprocessing time for simulated annealing is not reviewed in the table since the number of iterations were specified by hand. However, it was clear from experiments that it converged much slower than the preprocessing time for any other method.

An interesting observation is that in the time it takes to perform a single *Baseline* search, a trie can be both built and searched, using any of the algorithms in table 5.8, except for *Greedy* intersection variations and the alternative implementation of *Greedy*.

Algorithm	Construction Time
GlobalOrder - (ascending/descending frequency)	34 ms
Greedy	102 ms
Greedy - (alt. implementation)	173 ms
Group-Greedy	128 ms
Intersect-Avg-Greedy	7856 ms
Intersect-Max-Greedy	7989 ms

Table 5.8: The time required to preprocess the data using the different algorithms with cached data.

6

Conclusion

This thesis set out to optimize a weighted set containment algorithm and to investigate how preprocessing could be utilized to improve it. The study clearly shows that there are significant improvements that can be made with and without preprocessing.

The data suggests that a linear search strategy can be made significantly more efficient if implemented with a hash table. Furthermore, in a situation where the data set is not cached in RAM, there is much to gain from adopting a lazy fetch strategy where only the necessary amount of data is fetched.

It is evident from the results that preprocessing using set-tries can greatly increase the performance of the algorithm, at least in redundant data sets. Additionally, set-tries reduce the amount of memory required to store the data, which could prove important if there is not sufficient RAM to store all the data. An approach where set-tries are computed in advance and stored in a database was not investigated, but the reduced amount of memory required suggests that such an approach could prove to be very efficient, especially if combined with a lazy fetch strategy.

This thesis demonstrated the complexities of the minimum set-trie problem and proved it to be NP-complete. Additionally, a number of different algorithms for approximating a minimum set-trie were suggested and evaluated. Some of these algorithms performed notably better than the state of the art solution of using a frequency based global order, although this came at the cost of increased preprocessing time. None of the algorithms have been proven to guarantee an approximation of the optimal solution by a constant factor, but they appear to perform well in practice. It should be noted that we have made no claims as to whether such an algorithm exists and looking for such an algorithm or disproving its existence could serve as a future research path.

The results suggest that taking disqualification probabilities into account does not make any significant difference in the total search time of the algorithm. However, it is likely that a method using a disqualification probability based approach will perform well on a different data set. Hence, further research into such methods could prove lucrative.

Finally, there might be more relevant research done in the area of decision trees which we have not had the time to go through. A more in-depth examination of how decision trees are related to the optimal set-trie problem could potentially lead to better solutions.

Bibliography

[1] P. Bouros, N. Mamoulis, S. Ge, and M. Terrovitis, "Set containment join revisited," *Knowledge and Information Systems*, vol. 49, no. 1, pp. 375–402, 2016.

[2] I. Savnik, *Index Data Structure for Fast Subset and Superset Queries*, pp. 134–148. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[3] L. Liu and M. T. Zsu, *Encyclopedia of Database Systems*. Springer Publishing Company, Incorporated, 1st ed., 2009.

[4] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, pp. 490–499, Sept. 1960.

[5] R. Jampani and V. Pudi, *Using Prefix-Trees for Efficiently Computing Set Joins*, pp. 761–772. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[6] V. V. Vidal, *Applied Simulated Annealing*. Springer-Verlag, 1st ed., 1993.

[7] V. Horký, P. Libič, A. Steinhauser, and P. Tůma, "Dos and don'ts of conducting performance measurements in java," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, (New York, NY, USA), pp. 337–340, ACM, 2015.

[8] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *SIGPLAN Not.*, vol. 42, pp. 57–76, Oct. 2007.

[9] M. Rodriguez-Cancio, B. Combemale, and B. Baudry, "Automatic microbenchmark generation to prevent dead code elimination and constant folding," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, (New York, NY, USA), pp. 132–143, ACM, 2016.

[10] J. Clausen, "Branch and bound algorithms – principles and examples," 1999.

[11] I. Chikalov, *Average Time Complexity of Decision Trees*, vol. 21. Springer-Verlag Berlin Heidelberg, first ed., 2011.

[12] P. Slavík, "A tight analysis of the greedy algorithm for set cover," in *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, (New York, NY, USA), pp. 435–441, ACM, 1996.

[13] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *SCIENCE*, vol. 220, no. 4598, pp. 671–680, 1983.

A

Optimized Greedy Algorithm

Algorithm 4 Optimized Greedy Set-trie Build

```
1: procedure buildGreedySetTrie(parent, candidateSets)
2:     if not candidateSets.isEmpty then
3:         return
4:     chosenElement ←most common element among candidateSets
5:     globalElements ←elements that exists in all candidateSets
6:     if not globalElements.isEmpty then
7:         newNode ←null
8:         for all element in globalElements do
9:             newNode ← parent.addChild(element)
10:            parent ← newNode
11:        chosenCandidateSets ←[]
12:        for all set in candidateSets do
13:            set.removeAll(globalElements)
14:            if set.isEmpty then
15:                newNode.markAsEndNode
16:            else
17:                chosenCandidateSets.add(set)
18:        buildGreedySetTrie(newNode, chosenCandidateSets)
19:    else
20:        newNode ← parent.addChild(chosenElement)
21:        chosenCandidateSets ←[]
22:        remainingCandidateSets ←[]
23:        for all set in candidateSets do
24:            if set.contains(chosenElement) then
25:                set.remove(chosenElement)
26:                if set.isEmpty then
27:                    newNode.markAsEndNode
28:                else
29:                    chosenCandidateSets.add(set)
30:            else
31:                remainingCandidateSets.add(set)
32:        buildGreedySetTrie(newNode, chosenCandidateSets)
33:        buildGreedySetTrie(parent, remainingCandidateSets)
```