Substructure search optimizations

November 19, 2023

Abstract

Substructure search in chemical compound databases is a fundamental task in cheminformatics with critical implications for fields such as drug discovery, materials science, and toxicology. However, the increasing size and complexity of chemical databases have rendered traditional search algorithms ineffective, exacerbating the need for scalable solutions. This paper introduces a novel approach to enhance the efficiency of substructure search, moving beyond the traditional full-enumeration methods. Our strategy employs a unique index structure: a tree that segments the molecular data set into clusters based on the presence or absence of certain features. This innovative indexing mechanism is inspired by the binary Ball-Tree concept and demonstrates superior performance over exhaustive search methods, leading to significant acceleration in the initial filtering process. Comparative analysis with the existing Bingo algorithm reveals the efficiency and versatility of our approach. Although the current implementation does not affect the verification stage, it has the potential to reduce false positive rates. Our method offers a promising avenue for future research, meeting the growing demand for fast and accurate substructure search in increasingly large chemical databases.

1 Introduction

Substructure search in chemical compound databases is a vital task in cheminformatics, underpinning broad applications in drug discovery, materials science, and toxicology. The objective is to identify all molecules in a database that contain a given query substructure, which typically corresponds to a specific chemical motif or functional group. This search has been a cornerstone in understanding the influence of specific substructures on the biological activity, physicochemical properties, and reactivity of a compound, a concept recognized for decades [1].

Historically, computer-based substructure search started with pioneers like Ledley and colleagues who developed the Chemical Substructure Search (CSS) algorithm in the 1960s and 1970s [6]. CSS paved the way for modern algorithms by introducing the idea of using a computer to aid in chemical analysis. This algorithm employed a graph-based approach to identify specific substructures in chemical compounds. Further advancements in the field came with the development of the Simplified Molecular Input Line Entry System (SMILES) notation by Weininger in the 1980s [12], which provided a simple, linear representation of molecular structures as strings.

The search for a structure fundamentally relies on the solution of the subgraph isomorphism problem, a problem known to be NP-complete [11]. Due to its high computational complexity, numerous algorithms and heuristics have been devised to accelerate the search process. Among these, the Filter-and-Verification paradigm has been a prevalent approach, involving an initial filtering step to quickly eliminate unsuitable candidate graphs, and a more computationally intensive verification step to confirm the presence of the query substructure in the remaining candidates [4, 10]. Over time, graph-based subgraph isomorphism algorithms, such as the Ullmann algorithm [11] and the VF2 algorithm [4], have emerged as more efficient and scalable solutions for substructure search in large chemical databases.

In addition to these, frequent subgraph mining algorithms like gSpan [13], FFSM [?], and Gaston [7] have proven valuable in identifying frequently occurring substructures in large sets of chemical compounds. These approaches are particularly beneficial for applications such as structure-activity relationship (SAR) analysis and molecular classification.

Efficient filtering techniques often involve the use of binary and quantitative features, or fingerprints, to represent molecular structures. These fingerprints facilitate the rapid elimination of graphs that do not contain the specific features required by the query subgraph, thereby speeding up the substructure search process [2, 5].

However, as the number of known molecules and the size of chemical databases have grown significantly, traditional approaches, which often require a full or nearly full enumeration of candidates, have become increasingly challenging to implement efficiently. This development underscores the need for more scalable solutions. The complexity is not just computational but also involves handling increasingly large data sets that cannot be efficiently processed using traditional methods.

Our work introduces a unique approach to mitigating these challenges. While in certain cases the algorithm may resort to exhaustive enumeration, in most scenarios it employs a more sophisticated strategy, transcending the conventional full enumeration paradigm. Instead, we introduce a unique index structure: a tree that segments the molecular dataset into clusters based on the presence or absence of features. Inspired by the binary Ball-Tree concept [8, 3], this structure demonstrates superior performance over exhaustive search on average, leading to a significant acceleration in the filtering process.

We provide a comparative analysis of our method based on Bingo [9] with the original Bingo algorithm, highlighting key differences.

Although Bingo uses advanced filtering effectively, it relies on exhaustive search in a chemical space that continually expands. On the contrary, our approach departs from exhaustive search and places an existing molecular fingerprint into a tree structure, rather than a conventional relational database.

While our current version does not impact the verification stage, it speeds up the filtering stage. By introducing this innovative structure, we aim to cater to the growing scale of chemical databases and the escalating demand for efficient and scalable search solutions. Our approach offers potential for future research and application in the quest for more efficient and accurate substructure search techniques.

2 Algorithm description

2.1 Notation and main idea

The objective of our research is to facilitate the identification of specific substructures within molecules from a database $\mathcal{M}$. For this, we utilize the concept of a "fingerprint", a binary string of constant length fl, corresponding to each molecule. To perform this mapping, we define a function $\mathsf{fp} : \mathcal{M} \to \mathcal{F}$ that takes a molecule from the set $\mathcal{M}$ and produces its corresponding fingerprint in the set $\mathcal{F}$.

To make the substructure search process more efficient, we propose organizing these fingerprints in a binary search tree, denoted $\mathbb{T}$. The tree is binary and complete, having a specific depth d.

In this tree, the root, left and right subtrees of a node `v` are represented as $\mathbb{T}$`.root`, `v.left`, and `v.right`, respectively. Each node also has a set of all leaves in its subtree, denoted as `v.leaves`. Each leaf ℓ in the tree $\mathbb{T}$ holds a set ℓ`.set` of fingerprints. A unique concept of our approach is the centroid, `v.centroid`, recorded at each node v. The centroid is defined as a fingerprint F for which $F[i] = 1$ if and only if there exists another fingerprint F' in the subtree of `v` such that $F'[i] = 1$. This is represented as

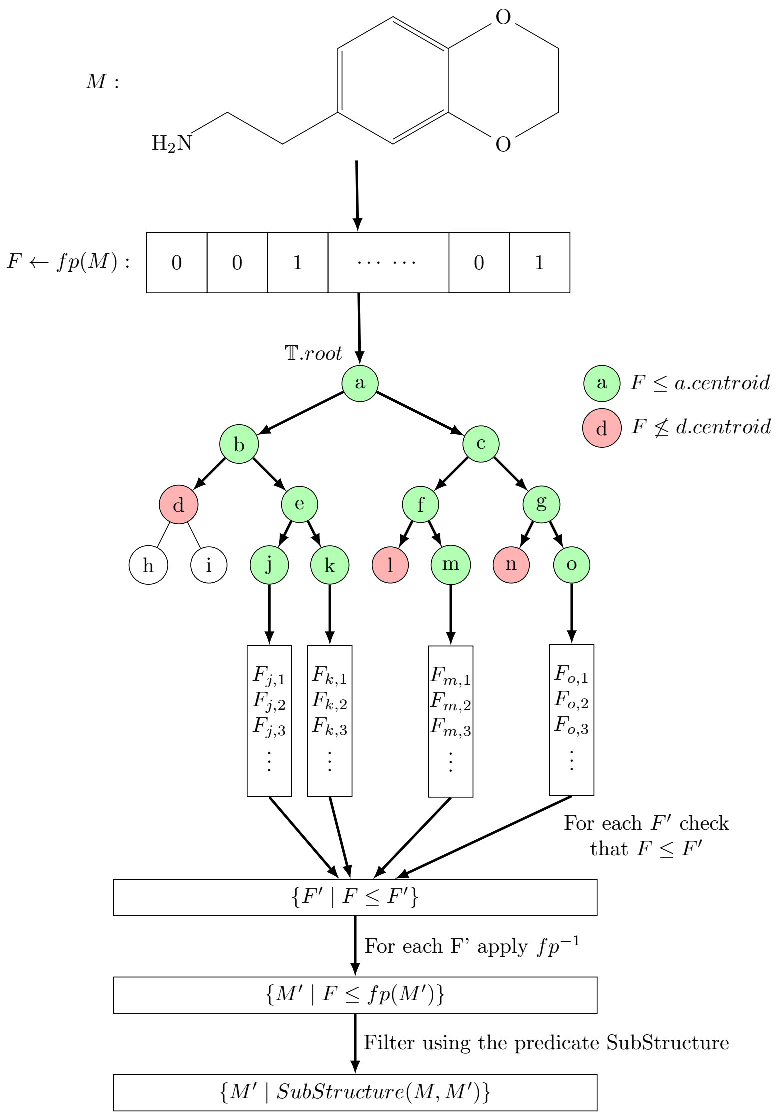
$$\texttt{v.centroid} = \bigvee_{\ell \in \texttt{v.leaves}} \bigvee_{F \in \ell.\texttt{set}} F.$$

This concept of the centroid is inspired by BallTree literature.

Our search process is designed to locate all fingerprints F' in the set $\mathcal{F}$ where F is a submask of F'. This search is based on the relation $F_1 \leq F_2$ for the fingerprints F_1, F_2 that holds true if and only if for every $i \in 1, 2, \ldots, \mathsf{fl}$, $F_1[i] \leq F_2[i]$.

The search starts from the root and recursively descends into both subtrees. Note that here we can improve the performance by parallelizing this step to explore both subtrees simultaneously. We stop the recursive descent if we reach a node `v` where $F \not\leq \texttt{v.centroid}$. Conversely, if we reach a leaf ℓ and $F \leq \ell.\texttt{centroid}$, we add to $\mathcal{F}_M$ the set $\{F' \in \ell.\texttt{set} \mid \mathsf{fp}(M) \leq F'\}$.

Following the generation of $\mathcal{F}_M$, the next phase involves examining each M' in $\bigcup_{F' \in \mathcal{F}_M} \mathsf{fp}^{-1}(F')$. The objective is to determine whether each M' is a substructure of M. This determination is made by employing external algorithms to verify the predicate $\texttt{SubStructure}(M', M)$, which is true if and only if M' is a substructure of M.


M :
H_2N
O
O
$F \leftarrow fp(M)$:
0
0
1
$\cdots\ \cdots$
0
1
$\mathbb{T}.root$
a
$F \leq a.centroid$
d
$F \not\leq d.centroid$
b
c
d
e
f
g
h
i
j
k
l
m
n
o
$F_{j,1}$ $F_{j,2}$ $F_{j,3}$ $\vdots$
$F_{k,1}$ $F_{k,2}$ $F_{k,3}$ $\vdots$
$F_{m,1}$ $F_{m,2}$ $F_{m,3}$ $\vdots$
$F_{o,1}$ $F_{o,2}$ $F_{o,3}$ $\vdots$
For each F' check that $F \leq F'$
$\{F' \mid F \leq F'\}$
For each F' apply fp^{-1}
$\{M' \mid F \leq fp(M')\}$
Filter using the predicate SubStructure
$\{M' \mid SubStructure(M, M')\}$

Further details on the BallTree and the utilization of the tree in the substructure search process will be provided in the subsequent sections.

The pseudocode for the fingerprint search function in the tree is described in Algorithm 1. The pseudocode for the function that searches for superstructures of a given molecule is described in Algorithm 2.

Algorithm 1 Searching for all matching fingerprints in a subtree

```
Require: v is a tree vertex, $F$ is a fingerprint
Ensure: $\{F' \in \bigcup_{\ell \in \texttt{v.leaves}} \ell.\texttt{set} \mid F \leq F'\}$
 1: procedure FindInSubtree(v, $F$)
 2:     if $F \nleq \texttt{v.centroid}$ then
 3:         return $\varnothing$
 4:     else if v is leaf then
 5:         return $\{F' \in \texttt{v.set} \mid F \leq F'\}$
 6:     else
 7:         left ← FindInSubtree(v.left, $F$)
 8:         right ← FindInSubtree(v.right, $F$)
 9:         return Concatenate(left, right)
10:     end if
11: end procedure
```

Algorithm 2 Searching for all superstructures of a given molecule

```
Require: $M$ is a molecule
Ensure: $\{M' \in \mathcal{M} \mid \texttt{SubStructure}(M, M')\}$
 1: procedure FindMetaStructures($M$)
 2:     $F \leftarrow \mathsf{fp}(M)$
 3:     $F_M \leftarrow$ FindInSubtree($\mathbb{T}$.root, $F$)
 4:     return $\{M' \in \bigcup_{F' \in F_M} \mathsf{fp}^{-1}(F') \mid \textsc{SubStructure}(M, M')\}$
 5: end procedure
```

2.2 Building the tree

To start, let us create a trivial tree with a single node, denoted as $\mathbb{T}$.`root`. Assign $\mathbb{T}$.`root.set` $= \mathcal{F}$. Next, we will inductively split the leaves of the tree into two parts, thereby adding new nodes to the tree.

More formally, for each leaf node ℓ of the tree, we will divide ℓ.`set` using a specific function called SplitFingerprints: $\mathcal{F}_l, \mathcal{F}_r \leftarrow$ SplitFingerprints(ℓ.`set`) ($\mathcal{F}_l \sqcup \mathcal{F}_r = \ell$.`set`). Next, we will recursively build trees for ℓ.`left`, ℓ.`right` using the sets $\mathcal{F}_l, \mathcal{F}_r$.

We will continue to split the leaves in this manner until $\mathbb{T}$ becomes a full binary tree with depth d. The pseudocode for the algorithm described above can be found in 3.

Algorithm 3 Building the tree

```
Require: $\mathcal{F}$ is the set of all fingerprints, $d$ is the depth of the tree
Ensure: $\mathbb{T}$ is the BallTree for the superstructure fingerprint search
1: procedure BuildTree($\mathcal{F}, d$)
2:     v $\leftarrow$ new node
3:     if $d = 1$ then
4:         v.set $\leftarrow \mathcal{F}$
5:         v.centroid $\leftarrow \bigvee_{F \in \mathcal{F}} F$
6:         return v
7:     else
8:         $\mathcal{F}_l, \mathcal{F}_r \leftarrow$ SplitFingerprints($\mathcal{F}$)
9:         v.left $\leftarrow$ BuildTree($\mathcal{F}_l, d - 1$)
10:        v.right $\leftarrow$ BuildTree($\mathcal{F}_r, d - 1$)
11:        v.centroid $\leftarrow$ v.left.centroid $\vee$ v.right.centroid
12:        return v
13:    end if
14: end procedure
```

Algorithm 4 Algorithm for splitting fingerprints in parts during tree construction

```
Require: set $\mathcal{F}$ of fingerprints to be split
Ensure: the split $\mathcal{F}_l, \mathcal{F}_r$ of the set $\mathcal{F}$
1: procedure SplitFingerprints($\mathcal{F}$)
2:     $j \leftarrow \arg\min_i \{||\mathcal{F}| - 2k| \mid k = \#\{F \in \mathcal{F} \mid F_i = 1\}\}$
3:     $\mathcal{F}_l \leftarrow \{F \in \mathcal{F} \mid F[j] = 0\}$
4:     $\mathcal{F}_r \leftarrow \{F \in \mathcal{F} \mid F[j] = 1\}$
5:     if $|\mathcal{F}_l| > \lfloor \frac{n}{2} \rfloor$ then
6:         $\mathcal{F}_r \leftarrow \mathcal{F}_r \cup$ TakeLastElements($\mathcal{F}_l, |\mathcal{F}_l| - \lfloor \frac{n}{2} \rfloor$)
7:         $\mathcal{F}_l \leftarrow$ DropLastElements($\mathcal{F}_l, |\mathcal{F}_l| - \lfloor \frac{n}{2} \rfloor$)
8:     else if $|\mathcal{F}_r| > \lceil \frac{n}{2} \rceil$ then
9:         $\mathcal{F}_l \leftarrow \mathcal{F}_l \cup$ TakeLastElements($\mathcal{F}_r, |\mathcal{F}_r| - \lceil \frac{n}{2} \rceil$)
10:        $\mathcal{F}_r \leftarrow$ DropLastElements($\mathcal{F}_r, |\mathcal{F}_r| - \lceil \frac{n}{2} \rceil$)
11:    end if
12:    return $\mathcal{F}_l$, $\mathcal{F}_r$
13: end procedure
```

We want to perform the splits in such a way that, on average, the search often prunes branches during the traversal. That is, the **if** statement in line 2 of the algorithm 1 should be executed frequently. Let us discuss the function SplitFingerprints in more detail.

Initially, one might consider selecting a specific bit j and assigning all fingerprints F such that $F[j] = 0$ to the left subtree, and those with $F[j] = 1$ to the right subtree. In this case, when searching for superstructures of the fingerprint F', if $F'[j] = 1$, the entire left subtree would be cropped. However, in practice, this approach leads to significant differences between the left and right parts after a few splits, making it difficult to create a deep and balanced tree. Unfortunately, a shallow or unbalanced tree does not offer substantial improvements over a full search, as it barely eliminates any search branches.

Therefore, we suggest the following method: we will still select the bit as mentioned above, but we will divide the fingerprints in a way that ensures the sizes of the resulting partitions match. For instance, if the optimal division of n fingerprints yields parts with sizes $n_0, n_1 (n_0 < n_1 \ \wedge \ n_0 + n_1 = n)$, then all values with zero will be assigned to the left partition, while the values with one will be distributed to achieve final left and right partition sizes of $\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil$ respectively. If $n_0 > n_1$, we will proceed symmetrically. The algorithm for the SplitFingerprints function can be found in the pseudocode 4.

3 Benchmarks

In this study, we have performed a comprehensive benchmarking to assess the performance of our algorithm, which is an extension of the Bingo fingerprinting system, compared to the established index, namely Bingo [9]. Our benchmarking process was performed under the following conditions:

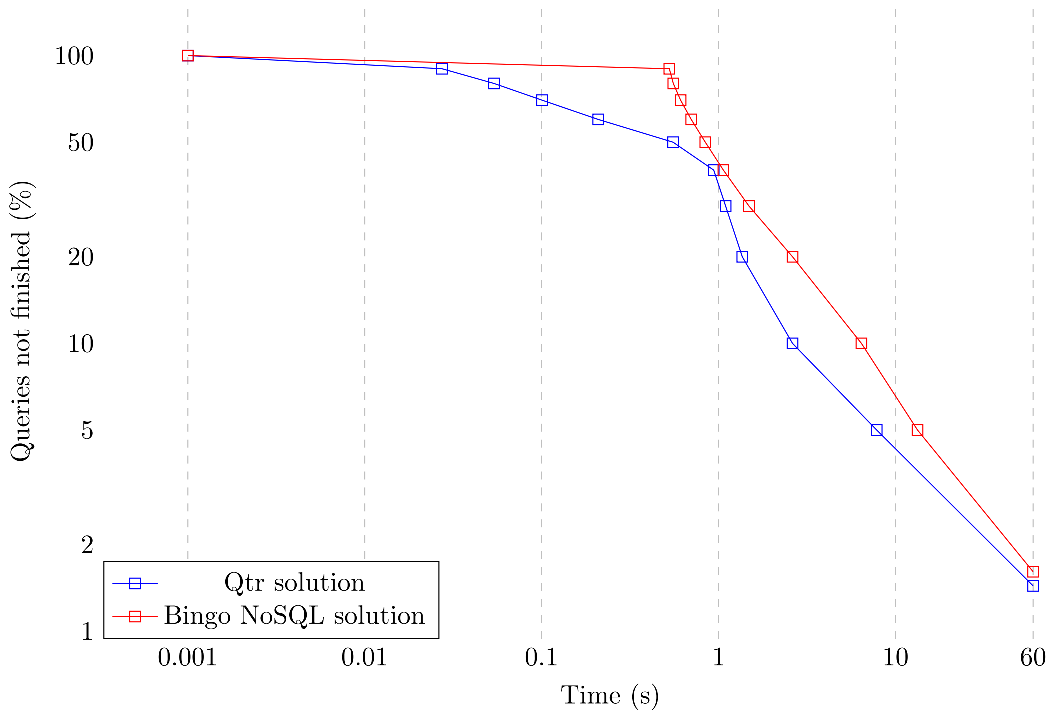
- OS: Ubuntu 22.04
- Processor: Intel Xeon E5-2686 v4 (Broadwell)
- Clock speed: 2.7 GHz
- RAM: 120 GB

The query dataset used for the benchmarking was retrieved from `https://hg.sr.ht/~dalke/sqc/browse?rev=tip`, which contains 3488 relevant queries for the substructure search. Ten queries were excluded due to various issues, resulting in a final set of 3478 compounds.

For a single-threaded in-memory execution, our algorithm demonstrates competitive performance, and it also shows the potential for parallelization, exhibiting substantial improvements when executed on 16 threads in memory.

The table below summarizes these benchmark timings, providing a clear comparison between our Qtr algorithm and the Bingo algorithm.

%	Qtr Algorithm, single-threaded, in-memory	Bingo NoSQL, single-threaded
10%	0.0273381	0.526846
20%	0.053802	0.554869
30%	0.100528	0.610074
40%	0.208735	0.700541
50%	0.553334	0.841574
60%	0.938981	1.06477
70%	1.09632	1.48609
80%	1.36175	2.61958
90%	2.61875	6.42211
95%	7.83572	13.3279
$\leq$ 60 seconds:	98.56%	98.39%



This detailed analysis offers valuable information about the performance and potential scalability of our algorithm, especially when it comes to parallel computing. Waiting for the benchmarks for the parallel version

4 Further Development

Fingerprints currently form the basis of our algorithm, but they do have certain limitations which do not make them the ideal fit for our tree-based approach.

Firstly, the condensed nature of the fingerprint is aimed at ensuring efficient computation, which often leads to grouping together several characteristics. For instance, a single attribute in a fingerprint often encapsulates multiple individual elements because these isolated items, while lacking substantial filtering power across the entire dataset, might be relevant for specific subsets. However, the fingerprint structure does not account for such instances. On the contrary, our approach could accommodate more complex functions, even if they operate slower than traditional filtering methods, for example, using a fingerprint variant that does not amalgamate different elements.

Secondly, fingerprints are designed to provide a universal filter throughout the dataset. This results in a significantly reduced set attributes applicable to the entire database. For example, Bingo utilizes 2584 attributes, which intuitively seem insufficient to capture all the peculiarities of a 113M-sized molecule dataset. Even a substantially enlarged fingerprint variant would not be able to cover all exceptional cases. In contrast, our approach, by dealing with subsets, can extract a unique characteristic for a tree node relevant to the set in the given subtree, thus allowing for much more effective coverage of the existing data nuances.

As a result, a potential enhancement of our algorithm might involve the use of a specific attribute in each tree node. Depending on its presence or absence, the search continues in both subtrees or only in the right subtree. This attribute would be chosen in advance to approximately bisect the set in the subtree. A leaf would contain several characteristics that would be examined when filtering elements from the leaf.

Using the method described above, we could potentially improve the false-positive rate, as the selected attributes would be relevant to the examined subsets. Moreover, these attributes could be utilized during verification, possibly resulting in substantial improvements in the verification stage speed, thanks to the relevance of these attributes to the molecule subsets.

5 Conclusion

The current version of our approach can serve as an extension of a fingerprint, enhancing filtering speed by avoiding exhaustive enumeration. Moreover, the tree's ability to cluster molecules enables a more detailed examination of cluster-specific attributes, an aspect that existing algorithms struggle with, as they aim to find optimal ways to generalize across the entire dataset. Therefore, our approach could potentially be used in the future to improve both the false-positive rate and the verification speed.

6 Availability of Data and Materials

The source code and datasets used in this study are openly available in the following repository:

- **Repository Name:** Quantori QTR Fingerprint
- **GitHub Link:** `https://github.com/quantori/qtr-fingerprint`

The repository contains all the necessary source code, data, and instructions required to replicate the experiments and analyses presented in this paper. This includes scripts for data preprocessing, model training, evaluation, and any additional utilities used in the study.

For further inquiries regarding the data and code, or for assistance with replication efforts, please contact the corresponding author or raise an issue in the GitHub repository.

References

[1] J. M. Barnard. Substructure searching methods: Old and new. *Journal of Chemical Information and Computer Sciences*, 33(4):532–538, 1993.

[2] F. Bonchi, R. Perego, F. Silvestri, H. Vahabi, and R. Venturini. Exemplar queries: Give me an example of what you need. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, pages 2097–2100. ACM, 2011.

[3] K. L. Clarkson. Nearest-neighbor searching and metric space dimensions. *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, 2(2006):15–59, 1994.

[4] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[5] K. Klein, M. Werth, and N. M. Kriege. Efficient subgraph mining in cheminformatics and elsewhere. *Informatik Spektrum*, 37(1):9–16, 2014.

[6] R. S. Ledley, L. B. Lusted, and J. D. Schulman. Computer-based medical decision making: The use of computers for the identification of chemical substructures. *Science*, 146(3647):1043–1045, 1964.

[7] S. Nijssen and J. N. Kok. A quickstart in frequent structure mining can make a difference. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 647–652. ACM, 2004.

[8] S. M. Omohundro. Five balltree construction algorithms. Technical report, International Computer Science Institute, 1989.

[9] D. Pavlov, M. Rybalkin, and B. Karulin. Bingo from scitouch llc: chemistry cartridge for oracle database. *Journal of Cheminformatics*, 2(1):F1, May 2010.

[10] D. Shasha, J. T. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the Twenty-First ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 39–52, 2002.

[11] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[12] D. Weininger. Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of Chemical Information and Computer Sciences*, 28(1):31–36, 1988.

[13] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, pages 721–724. IEEE, 2002.