An Implementation of Hypersuccinct Trees

Christopher Pack, Alexander Auras, Nathanael Stöhr, Charbono Lonyem Tegomo

December 8, 2021

Abstract

This document describes an implementation of Hypersuccinct Tree encodings in C++.

Contents

1 Introduction

...

2 The tree covering Algorithm

Farzan Munro und Universal Succinct - Theorie

2.1 Functions and Queries of the Hypersuccinct Tree

Explains all Queries and roughly their theoretical implementation [1] [2]

2.2 Efficiencies

Space Efficiency
Algorithm Efficiencies

3 Our Implementation

Why C++?

3.1 Goals of the Implementation

Proof of concept
Workable library
Basic functions such as implemented queries or file writing

3.2 External Libraries

We use four external libraries in our project. The first one, called Irrxml, serves an implementation to read efficiently from an xml file. We use this library to generate a basic tree structure from xml files which is then used for our encoding.
We make use of a small library which provides an implementation of space efficient Fully Indexable Dictionaries [2], called succinct-bv. This library is very lightweight and has been modified personally by us to suit our needs.
To increase the performance of our encoding process (see section 4.2), we utilize a multithreading library. This library is called thread-pool, an was created by [3] to provide an efficient multithreading library.
Lastly, we utilise the googletest library for unit tests.

4 Implementation in Depth

Amount of Classes,
Structures of classes.

4.1 The Hypersucccinct Tree class

The Hypersuccinct Tree class implements the entire hypersuccinct tree encoding. It consists of a struct representing minitrees and one representing the lookup table entries. Both of these structs use plenty of bitvectors to store relevant data for queries.

Hypersuccinct Nodes:
Nodes within the Hypersuccinct Tree are identified as triplets, where the first value denotes the Minitree, the second the Microtree and the third the Node number within the Microtree.

4.1.1 The FID and identifying trees

FIDs represent the interconnections between trees at their roots, their type 1 and type 2 connections [1], together with a Type Vector that shows which kind of connection the tree has. Due to the way they are being generated, the FIDs do not share indices with their respective trees (i.e. it is possible for the Minitree with $id = 7$ to be in the FID with $id = 5$). This necessitates an index conversion whenever the FID or the type vector is used. Doing this conversion at runtime would not be possible within $O(1)$, since it would be necessary to go through each FID and count the trees contained within it, for both type 0 and type 1 connections. This is further complicated by taking into account the fact that each tree without dummies is actually accounted in two different FIDs. Firstly in the FID of its own root as a type 0 connection (Top FID), and secondly in the FID of its parent as type 1 (Low FID). Trees below dummies however have no low FID, as they are not directly below the root of their parent tree. Therefore

Vector indices for Trees								
Minitree Number	0	1	2	3	4	5	6	7
Minitree Top FID	0	1	2	3	4	5	6	6
Minitree Low FID	-	0	0	2	2	3	-	-

Figure 1: Indices for Minitrees

we implement a two way index conversion by directly pointing to the type 0 and type 1 FID index for each tree, and pointing to the first type 0 and type 1 tree for each FID. Since this is $O(4 \times |Minitrees|) + O(4 \times |Microtrees|)$, it is still within our required space efficiency. Dummies on the other hand use direct pointers to their children.

Bitvector	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8
Minitree FID 0	1	1	0	1	0	0	1	1
Minitree Typevector 0	0	1	-	0	-	-	1	0
Minitree Numbers	0	3	0	1	1	1	5	2
Minitree FID 1	1	0	1	1	0	1	1	1
Minitree Typevector 1	0	-	1	1	-	1	1	0
Minitree Numbers	3	3	6	7	3	8	9	4

Figure 2: FID, Typevector and corresponding trees

The second issue arises when trying to analyze the FID and Typevectors of the Hypersuccinct Tree. Identifying which tree is at which position within even a single FID is not trivial. As demonstrated in *Figure 2*, the second low tree in the first FID is 5, because the FID of tree 3 contains tree 4 already as a top tree, meaning correctly identifying Bit 7 as tree number 5 requires analysis of the FID of tree number 3. Additionally, it is important to note that the FID with the ID of 1 starts with the tree numbered 3, which can only be known if all previous FIDs are also analyzed. Both of these analyzations exceed our time requirement of $O(1)$. Finding the first trees of any FID is clearly recursive starting from FID 0, while correctly identfying low trees requires checking $|LowTrees|$ many additional FIDs. Therefore, the first top and low tree of any FID is pointed to directly. This solves both of our issues, as the Rank of the Typevector indicates which FID belongs to which respective lower tree, allowing us to just take their first top tree as a result in $O(1)$. Knowing which tree an FID starts with is trivial with such pointers.

4.1.2 The Hypersuccinct Tree Factory

The Hypersuccinct Tree Factory is tasked with generating a hypersuccinct tree from a generic tree. Per our implementation this class implements two functions, one to generate hypersuccinct trees from our own generic tree structure, Unordered Tree, and a second one to generate from an already encoded file reading.
Encoding a generic tree is rather complicated due to the large amount of query related bitvectors that need to be filled with their respective data, a task that is not easily done on the generic tree structure. The complexity is therefore at least $O(|Nodes|^3)$, as we need to iterate through each minitree, within we iterate through each microtree, within we iterate through each node, as generated with the Farzan-Munro Algorithm. To increase the performance of this function, we make use of the multithreading library by [3]. In contrast recreating a hypersuccinct tree from an encoded file is straightforward and easy. The data values from the file are just copied directly into their expected bitvector fields, which requires a complexity of $O(|Minitrees|) + O(|Lookuptableentries|)$.
The Hypersuccinct Tree Factory generates all data necessary for query execution in $O(1)$, therefore fills all vector fields in Hypersuccinct Tree. The function

is structured in loops. First, the Farzan Munro algorithm is executed on a base tree with the computed Minitree size of $\lceil log^2(|Nodes|)\rceil$. Right after this, the Minitree interconnections are handled and Minidummies are generated. Then, the Farzan Munro algorithm is used on all generated Minitrees to generate their respective Microtrees of size $\lceil log(\frac{|Nodes|}{8})\rceil$, and their interconnections are handled. At last, for each of these Microtrees, a Lookup table entry is generated if an entry of such a tree sturcture does not already exist.
Special Cases:
There are some special occurences that lead to some additional complexity.

IMAGES

The first issue originates from the handling of interconnections and the node numbering. Adding dummies in the Unordered Tree class does not retain the original structure of the nodes, especially when the Minidummy is located right after the Microdummy on the same level (*Case 1*). Generating a consistent node numbering requires to sort the nodes within each trees. For this an *enumerate* function is implemented that creates a consistent order for nodes within Mini and Microtrees.
Another issue is the identification of Minitree dummies within Microtrees. Minitree dummies are originally identified by their node number respective to the Minitree. However, due to the structure of Hypersuccinct nodes, it is necessary to translate the Minitree node numbers to Microtrees and Microtree node numbers.

4.2 Query structures and differences

Not all queries represented in the paper are implemented. This is in part due to the simplicity of this implementation, but also because the original paper is rather shortworded when describing the implementation of each query, or simply points towards other papers related to this topic, which were not part of this implementation. Due to this there are some minor or notable differences to the paper at hand.

4.2.1 Simple Queries

As simple Queries we identify those that only require a single bitvector per level of abstraction. This means that these queries need one bitvector for values of each Minitree, one for each Microtree within the Minitrees and one for each lookuptable entry. As a result these queries all follow a similar structure, first handling single nodes, then generalizing the result to Microtrees and at last taking Minitrees into account to arrive at the correct result with an $O(1)$ time complexity.
These queries are:

1) *getParent*

2) *degree*

3) *subtreeSize*

4) *depth*

5) *height*

6) *leftmostLeaf*

7) *rightmostLeaf*

8) *leafSize*

9) *levelSuccessor (Not implemented)*

10) *levelPredecessor (Not implemented)*

These queries are described as using one bitvector per abstraction in the paper. We therefore believe this implementation to be as described in the paper.
Depth is specifically easy, since the depth of a given MicroTree or MiniTree root is just the size of their FIDs, making extra data bitvectors for depth unnecessary on a Minitree and Microtree level. All other queries use their respective bitvectors to determine a result.
On Dummies:
These queries require to handle the dummy in query-specific ways. *Depth* is the simplest query, only adding values from Minitree depth, Microtree depth and Node depth. These do already ignore dummies in their data, so this query does not require any dummy handling. *Height* and *subtreeSize* make use of the *isDummyAncestor* helper queries to subtract the dummy from their calculations. *Degree*, *leftmostLeaf* and *rightmostLeaf* check whether or not the current node is the dummy, and then move along its pointer. *GetParent* simply checks if the result is a dummy, and then returns the parent of that dummy. This is still within $O(1)$, as there are at most two dummies (one Microdummy and one Minidummy) on top of one another.
Despite not being implemented, we can still make some observations about *levelSuccessor* and *levelPredecessor*. Both cannot use the FID to compute their results, as identifying a specific node in the FID is not possible in $O(1)$. We therefore believe that a bitvector pointing to either the FIDs position or directly to the successor/predecessor is necessary, therefore making both of these Simple Queries. For dummies both would use the same handling as getParent, as the value returned by the dummy is the correct one, whereas the dummy's child would be effectively one level too low.

4.2.2 Rank Queries

Rank Queries are all queries that return some sort of rank from the tree. These queries are more compicated than Simple Queries since they require more than one bitvector per abstraction. Rank queries specifically need two bitvectors for

the Minitree and Microtree abstractions, due to special cases. They also require use of the FID to identify these special cases.
The Rank Queries are:

1) *childRank*

2) *leafRank*

3) *nodeRank (Not implemented)*

The special cases are as follows.

Case 1 and Case 2 describe a situation in which a higher indexed Microtree has a lower rank than the current Microtree. To resolve this, we introduce an extended rank bitvector which notes the rank of the first child of our current Microtree within the Minitree. This resolves this case on a data level, the second issue is to actually read the value of the extended bitvector in the right cases. This is where we differentiate between the two cases. If this problem appears at the Minitree root, we can identify this case by using the FID, resulting in Case 2. If this problem appears further down in the Minitree we can compare values of Microtree and Node indices to identify Case 1. Case 3 describes a similar issue regarding the Minitree. Again, a Minitree with a higher index has a lower rank than our current Minitree. This can also be identified with the FID. The three cases are mutually exclusive since they describe a similar issue. Case 4 describes a tree of type 0 that is split by a tree of type 1 in between. This can be identified, since 0s in the FID must belong to the last type 0 tree, meaning after identifying the latest tree as being of type 1, one only needs to check if the index points to a 0 in the FID.
The structure of both queries is similar. First they calculate the result of only the node within the Microtree, using the values from the lookup table. Then, the result is generalized to a Microtree. Here, Case 1 is evaluated. At last, the result is generalized to the Minitree by analyzing key structures of the FID to identify Case 2, or Case 3. In *childRank*, the special cases immediately return the value, as this query only needs to take into account the singular values of the parent FID. For *leafRank*, a generalization to the entire tree is always necessary due to the result being sensitive to the entire tree topology.
Both implemented queries use *getParentForQuery* as a helper query to identify the direct parent of the current node. These queries only check if the current node is a child of a dummy and if that is the case, the dummy is used instead, as the dummy is at the correct position to calculate the correct result.
The special cases are not mentioned in the paper. The description of all Rank Queries are rather short and inconclusive.
It is for example correct that we use the FID to compute the childRank. However, we interpret the paper to say that computing the childRank is possible by just using the FID, without any additional data. This is not possible due to the identification of a node within the FID requiring $O(|TreesInFID|)$ time, which

violates our time complexity requirement. Our implementation instead uses the FID to identify the special cases that require additional information and then take that information from the additional bitvectors we implemented. Both implemented Rank Queries also need to take their direct ancestor into account, which is not mentioned for LeafRank. We believe our approach to be adequate, since both implemented queries fulfill the $O(1)$ time requirement without violating the space requirement. Additionally, both of our Rank Queries are very similar in their implementation offering easier understandability and openness, since adding *nodeRank* would most likely just require the exact same steps as taken in our approach to Rank Queries.

4.2.3 Unique Queries

There are queries that do not fit into a category. These are:

1) *child*

Child is simpler than rank queries, since the index provided by the query is directly related to an FID position, allowing an analysis of the node at that position in $O(1)$. It is important to note that identifying the correct child within the FID can point to a 0 value in the FID. This means that we need to identify Microtrees and singular nodes via the distance from the last 1 within the respective FID of the Mini and Microtrees. To implement this, we first just identify the correct Minitree and reduce the remaining index by the position of that Minitree in the FID. The process for identifying the Microtree is analogous to that, and at last we use the remaining index to select a specific node via the lookup table. If our starting node is already below the Minitree root, the part for identifying the right Minitree is skipped, same for Microtrees if we are below a Microtree root. Dummy-nodes are skipped, both if the starting node is a dummy, in which case child of the direct pointer of the dummy is used instead, or if the result node is a dummy, in which case the direct pointer of the dummy is returned instead.

4.2.4 Helper Queries

Instead of being identified by their algorithmic structure, these queries are identified by the fact that they are used by other queries to compute their results. These Helper Queries are:

1) *isDummyAncestorWithinMiniTree*

2) *isDummyAncestorWithinMicroTree*

3) *getParentForQuery*

Their implementation is similar to the implementation of Simple Queries.
GetParentForQuery is a special version of *getParent* that excludes handling for dummy nodes. This is necessary to identify special dummy cases in the queries that use this Helper Query (such as *child*). The *isDummyAncestor*

helper queries identify if a given node is an ancestor of a Minidummy or a Microdummy respectively. Both queries only work if their current node is actually in a Minitree or Microtree with a dummy. *IsDummyAncestorWithinMiniTree* has a similar structure to the simple queries, first using specific bitvector data that contains if the Microtree root is an ancestor of the dummy, then if the specific node is the ancestor. *IsDummyAncestorWithinMicroTree* is even simpler, only needing to determine the answer for specific nodes. Both queries are necessary to evaluate an entire tree structure in $O(1)$, since they prevent the necessity to analyze every node on the structure individually, which would be $> O(1)$.

4.2.5 Other Queries / Not implemented queries

The following queries are not implemented and are not similar enough to already implemented queries to use those as a template.

1) *nodeSelect (Not implemented)*
2) *levelAncestor (Not implemented)*
3) *lowestCommonAncestor (Not implemented)*
4) *distance (Not implemented)*
5) *levelLeaftmostNode (Not implemented)*
6) *levelRightmostNode (Not implemented)*

Moving multiple levels is not trivial with our encoding. Since this is supposed to work in $O(1)$ time we are not sure how to implement a level based query. Moving up one level at a time is $> O(1)$, but adding one bitvector per level for each Microtree violates the space efficiency requirement.
NodeSelect has to call to a mapping of basic node numbers to HstNodes. Again, saving such a map for each Microtree and Minitree violates our space efficiency requirement, but computing it violates the time efficiency requirement.

4.3 Other Classes

There are a variety of other classes within the library that contribute to the Hypersuccinct Tree functionality.

4.3.1 Bitvector Utils

Bitvector Utils is a basic class that provides utility functions for Bitvectors. Its main function is to typecast Bitvectors to Integers and back, which is possible in O(1) due to the limited size of Integers. (i.e. uint 32 has a maximum of 32 bits.)

4.3.2 HST Output

The HyperSuccinctTree Output class provides functions for writing and reading a Hypersuccinct Tree class. Therefore this class handles writing Hypersuccinct Trees to Files or reading Hypersuccinct Trees from Files. It also supports writing the entire tree into the console.

4.3.3 Unordered Tree

Unordered Tree is our basic tree implementation, to which an xml tree is parsed. Unordered Trees are then used to create Hypersuccinct Trees.

4.3.4 Precomputed Function and Cached Function

These two classes are used to improve the performance of *create* by either completely precomputing values for queries within the Unurdered Tree, or by caching already calculated values.

4.3.5 XML Reader

This class reads the actual xml file and creates an Unordered Tree from it.

4.3.6 Farzan Munro

The Farzan Munro class implements the algorithm as presented in [1]. It therefore implements *decomopose* and *greedilyPack*. It also orders the created Tree afterwards to guarantee a consistent order within the hypersuccinct tree.

5 Complexity Benchmarks?

This section is mainly to show the difference between normal and Huffman encoding.

5.1 Huffman encoding

Our tree offers to types of encoding for Microtrees. The first option encodes trees according to [1], and uses the Microtree's balanced parenthesis representation, while the other utilizes a huffman encoding for the Microtrees. For that, the Microtrees use their respective huffman codes in the representation, while their corresponding lookup table entries contain the balanced parenthesis representation for each code.
The huffman encoding has no impact on query performance, as finding a corresponding lookup table entry for a huffman code is possible in $O(1)$. Encoding trees with huffman has a significant impact on the space necessary for the hypersuccinct tree. The balanced parenthesis representation needs $2n$ many bits for a tree of size n.

5.2 Space Complexity concessions

While the encoding is efficient in size when saving a tree to a file, within the program, the amount of space required to run trees is a lot larger. This technically does not break the space requirement, simply adding a large constant factor to it, but this is still notable. We identify the following reasons for this discrepancy:

1) The implementation of *Bitvector* in $C++$ saves its data as *uint32_t*, which is always 32 bits long. This is a massive waste of space, since most values are much smaller than 32 bits.

2) The use of *vector<Bitvector>* causes overhead due to the pointer lists that *vector* uses.

3) The *succinct_bv* library does not meet the perfect space complextity of $log(\binom{n}{r}) + O(\frac{nloglogn}{logn})$, and instead has a complexity of $n + O(\frac{nloglogn}{logn})$.

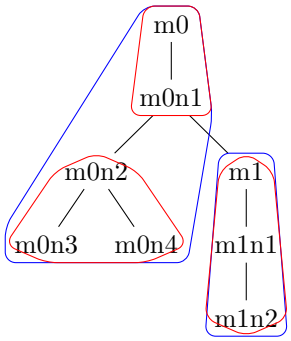
For *1), 2)*, we believe it to be possible to save the entire hypersuccinct tree in a single bitvector. It should be possible to use pointers to access different parts of this bitvector in $O(1)$.
For *3)*, it is obviously necessary to either use a library that offers the improved complexity, or to improve the current library.

References

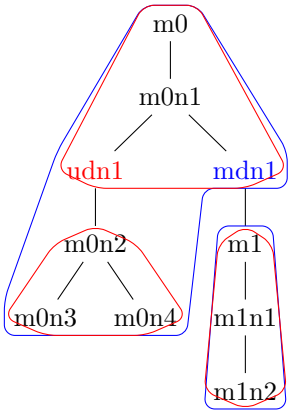
[1] Arash Farzan and J. Ian Munro. A uniformparadigm to succinctly encode various families of trees. *Springer Science+Business Media*, 2012.

[2] J. Ian Munro, Patrik K. Nicholson, Luisa Seelbach Benkner, and Sebastian Wild. Hypersuccinct trees – a universally compressed data structure for binary and ordinal trees. 2021.

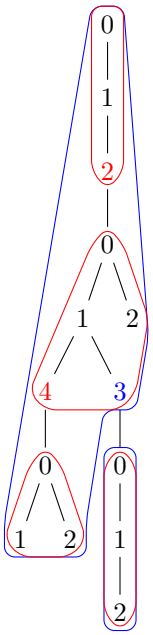
[3] Barak Shoshany. A c++17 thread pool for high-performance scientific computing. *arXiv:2105.00613*, 2021.



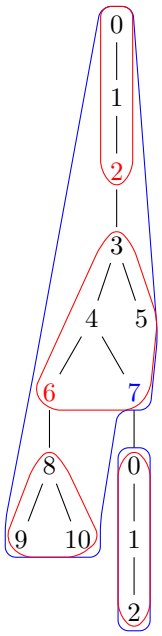
(a) Node order without Dummies



(b) Node order with Dummies

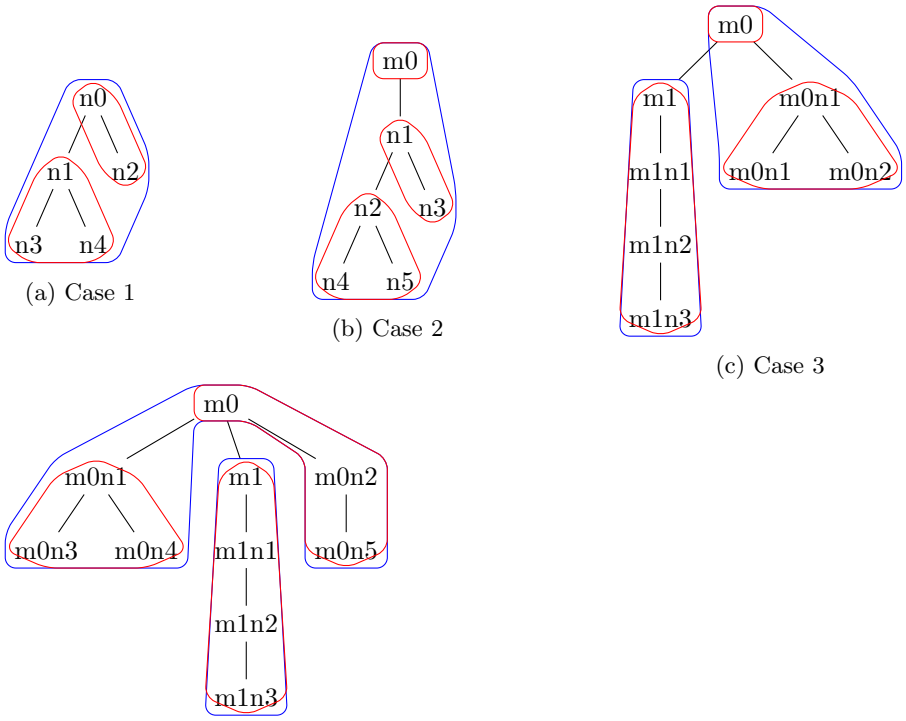


(c) Final Node numbers



(d) Minitree specific node numbers

Figure 3: Special Cases regarding node order



(a) Case 1

(b) Case 2

(c) Case 3

(d) Case 4

Figure 4: 4 Special Cases for Rank Queries

Bitvectors				
Bitvector	Case 1	Case 2	Case 3	Case 4
Minitree FID	1	1	11	110
Minitree Typevector	0	0	10	010
Microtree FID	11	11	irrelevant	irrelevant
Microtree Typevector	10	10	irrelevant	irrelevant

Figure 5: FID and Typevector values for special cases

Tree Name	Normal	Huffman	Huffman + Lookuptable
TreeNath.xml	52	23	24
TreeNath2.xml	484	256	266
TreeNath3.xml	5196	2695	2706
TreeNath4.xml	53369	31709	31732
TreeNath5.xml	583289	345005	345029
XMark2.xml	2004196	831572	831627
DBLP.xml	3690039	593804	593842

Figure 6: Space for normal encoding and huffman encoding