An Implementation of Hypersuccinct Trees

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

Lehrstuhl für theoretische Informatik
Universität Siegen

February 22, 2022

Outline

Table of Contents

Introduction

- We were tasked with creating an implementation of a hypersuccinct tree encoding, as described in [1].
- Additionally we add the possibility of huffman encoding for part of the tree, to provide an implementation of the encoding improvement mentioned in [2].
- Our result is a library that can encode trees in acceptable time, and is able to perform queries on those encoded trees in $O(1)$, while offering huffman encoding for their Microtrees.

Table of Contents

The tree covering Algorithm

[1] provide the decomposition algorithm, that splits a given tree into subtrees with the following properties:

- Given a splitting size m, all resulting trees are at most of size $2m$.
- The resulting subtrees at most share a single node, their root.
- This results in three types of connections between subtrees:
  - Subtrees share their roots.
  - Subtree roots are directly below other subtree roots.
  - Rarely, a subtree lies below a non-root node. In this case, we use a dummy node to represent this connection.

The tree covering Algorithm

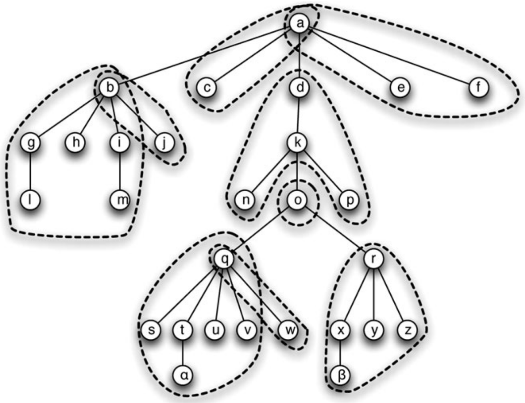

Figure: Example of Tree decomposition with $m = 5$ [1]

The tree covering Algorithm

To fully decompose a tree of size n for hypersuccinct encoding, [1] offers the following procedure:

- First decompose the entire tree into Minitrees with the size $\lceil lg^2 n \rceil$.
- Generate FIDs and Dummys for their interconnections.
- Then decompose each Minitree into Microtrees with the size $\lceil \frac{lgn}{8} \rceil$.
- Generate FIDs and Dummys for their interconnections.
- For each unique Microtree structure, create a lookuptable that saves relevant data for their individual nodes.

Queries

As mentioned in [1], additional data needs to be saved in order to execute queries without needing to decode hypersuccinct trees.

- Additional query data is saved for each level of abstraction (Minitrees, Microtrees, lookuptable).
- To execute some queries, navigation on the FID and the Typevector is required.
  - The FID and Typevector represent the interconnections between trees, there is one FID for each unique root.
  - The FID holds all children of its root and marks the first child of new trees.
  - The Typevector marks what type of connection the trees in its FID are.
  - A possible efficient implementation is described in [3], which is implemented in an external library [4].

Table of Contents

Hypersuccinct nodes and trees

- Hypersuccinct nodes are tripels that represent their Minitree, Microtree and Node within the Microtree.
- The Hypersuccinct Tree class implements the encoding and offers the query functions.
- The Hypersuccinct Factory handles creating a hypersuccinct tree from possible sources.

The Hypersuccinct Factory

- Encodes trees efficiently, making use of multiple threads with a multithreading library [5].
- Creates data for query execution.
- Can create hypersuccinct trees from an encoded file.

Minitrees and lookuptable entries

- Are part of the Hypersuccinct Tree class.
- Minitrees and lookuptable entries are structs that hold bitvectors.
- Minitrees hold all information for their Microtrees, all their query data and all their Microtree query data.
- Lookuptable entries hold information of their structure and a key for identification, as well as query data.

FIDs and specific issues

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 TV 0	0	1	-	0	-	-	1	0
Minitree Nr.	0	3	0	1	1	1	5	2
Minitree FID 1	1	0	1	1	0	1	1	1
Minitree TV 1	0	-	1	1	-	1	1	0
Minitree Nr.	3	3	6	7	3	8	9	4

Figure: Indices for Minitrees

FIDs and specific issues

- To solve the discrepancies with FID and tree indices:
  - Bitvectors that denote the first Type 0 (Top) and Type 1 (Low) tree of every FID.
  - Bitvectors the denote their Top and Low FIDs for each Tree.
  - This is done for both Mini- and Microtrees, so 8 bitvectors in total.
- To solve the issue with identifying the correct low trees:
  - When identifying trees, we always take the top tree of the FID that the low index points to.
  - This is possible since each FID points to its first low tree, which points to its own FID, and other low FIDs have incremental indices by construction.

Simple Queries

Simple queries are such that require only one bitvector per abstraction. Their structure simply moves from one abstraction level to the next one to answer the query.
These simple 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)*

Rank Queries

Rank queries are queries that return some sort of rank from the tree. These queries are all similar in structure, and need multiple bitvectors per level of abstraction, due to special cases. These rank queries are:

1) *childRank*
2) *leafRank*
3) *nodeRank (Not implemented)*

Special cases: Rank

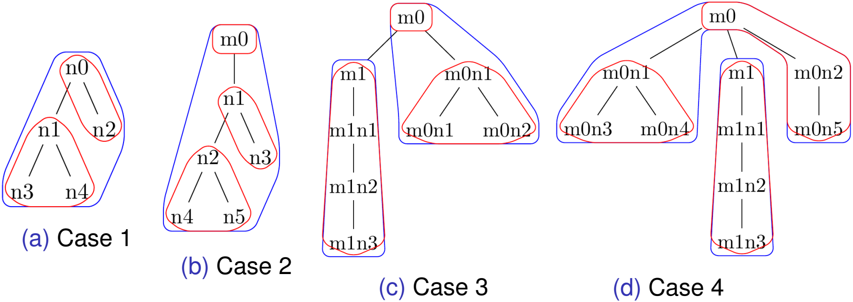

Figure: 4 Special Cases for Rank Queries

Special Cases: Rank

- We can use the FID to identify each special case. However, since rank queries do not provide an index in for the FID, the identification of the right position of the node in the FID is difficult, and we therefore need to provide an answer from the node indices alone.
- We need specific bitvectors that denote the rank of the first child of a tree for data to resolve these cases.
- We do not need two bitvectors in the lookuptable, since it saves data for every node.

Child

This is a very unique query, as it is simpler than rank queries, since the index provided points to direct positions on the FIDs:

- We can identify the correct node by moving through the Minitree FID, then the Microtree FID and then the lookuptable entry.
- Dummy nodes can easily be skipped both at the end and the beginning of the query.

Helper Queries

These queries are purely used within other queries to take on some repeat tasks:

1) *isDummyAncestorWithinMiniTree*
2) *isDummyAncestorWithinMicroTree*
3) *getParentForQuery*

Space complexity

Our implementation takes $4n + o(n)$ bits to encode a tree of size n:

1) The Microtree encoding takes $2n + o(n)$ bits, and huffman encoding reduces this further [2].
2) The FIDs takes $2n + o(n)$ bits in total.
3) All query data, all lookuptable entries, and all Typevectors and Dummys only take $o(n)$ bits each.

Table of Contents

Query tests

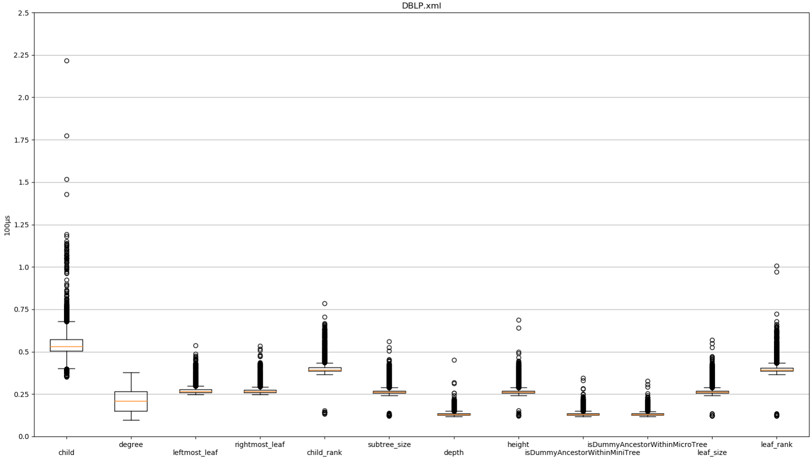

Figure: Runtime of implemented queries

Query tests

- All queries except *child* maintain an anverage runtime below 100μs.
- The runtime of *child* actually increases with larger trees. We identified the reason for this being a simple *getMinitree* function, which is also used in multiple other queries.

create, writing and reading

- The encoding process in *create* has been optimized as much as we think possible.
  - While the *farzan-munro algorithm* cannot be optimized with multithreading, the creation of Microtrees are individual tasks that can be parallelized.
  - Adding huffman encoding decreases efficiency slightly.
- Reading and writing files is straightforward and therefore much more efficient.
  - Writing just pushes every vector with Elias-Gamma encoding into a file.
  - Reading just decodes the vectors from the file. There is no handling of badly formatted vectors.

create, writing and reading

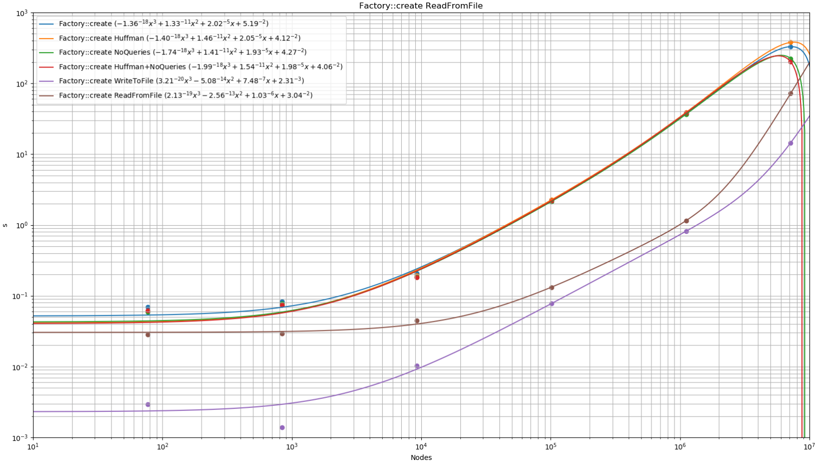

Figure: Runtime of create, reading and writing

Space and Huffman

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

Figure: Space for normal encoding and huffman encoding in byte

- Effective space reduction.
- XMark2 is larger than DBLP with huffman due to having more evenly distributed tree structures.

Table of Contents

– Demonstration of program –

Conclusion

- We have created a library that can encode trees succinctly.
- Our tree encoding is space efficient and allows us to execute various queries in $O(1)$.
- We offer huffman encoding for our tree, which saves space.
- The encoding process is optimized, can encode trees with more than 7 million nodes in less than 6 minutes.

References I

A. Farzan and J. I. Munro, “A uniform paradigm to succinctly encode various families of trees,” *Algorithmica*, vol. 68, no. 1, pp. 16–40, 2014. [Online]. Available: https://doi.org/10.1007/s00453-012-9664-0

J. I. Munro, P. K. Nicholson, L. S. Benkner, and S. Wild, “Hypersuccinct trees - new universal tree source codes for optimal compressed tree data structures and range minima,” in *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, ser. LIPIcs, P. Mutzel, R. Pagh, and G. Herman, Eds., vol. 204. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 70:1–70:18. [Online]. Available: https://doi.org/10.4230/LIPIcs.ESA.2021.70

References II

- R. Raman, V. Raman, and S. R. Satti, "Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets," *ACM Trans. Algorithms*, vol. 3, no. 4, p. 43, 2007. [Online]. Available: https://doi.org/10.1145/1290672.1290680
- "succinctbv," 2021, last accessed on 17 January 2022. [Online]. Available: https://github.com/ChristopherPack/succinct-bit-vector
- B. Shoshany, "A c++17 thread pool for high-performance scientific computing," *arXiv e-prints*, May 2021. [Online]. Available: https://arxiv.org/abs/2105.00613