On Dynamic Succinct Graph Representations

Miguel E. Coimbra, Alexandre P. Francisco, Luís M. S. Russo, Guillermo de Bernardo, Susana Ladra, Gonzalo Navarro

inesc id lisboa

citic

enxenio

Millennium Institute Foundational Research on Data

TÉCNICO LISBOA

UNIVERSIDADE DA CORUÑA

UNIVERSIDAD DE CHILE

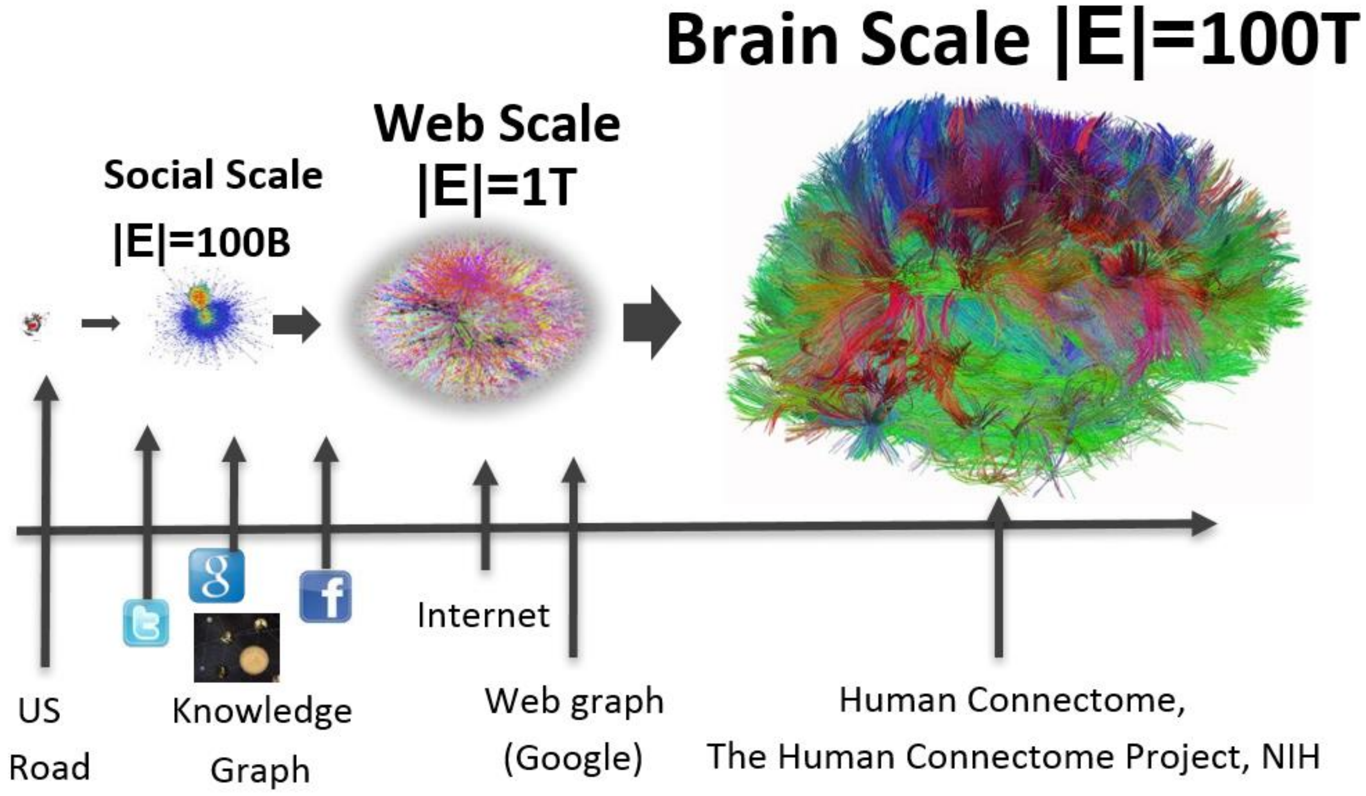
This research has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie [grant agreement No 690941]:

Bioinformatics and Information Retrieval Data Structures Analysis and Design (BIRDS)

Background

Importance of graphs, compression and k^2-trees

Graphs abound in size and types

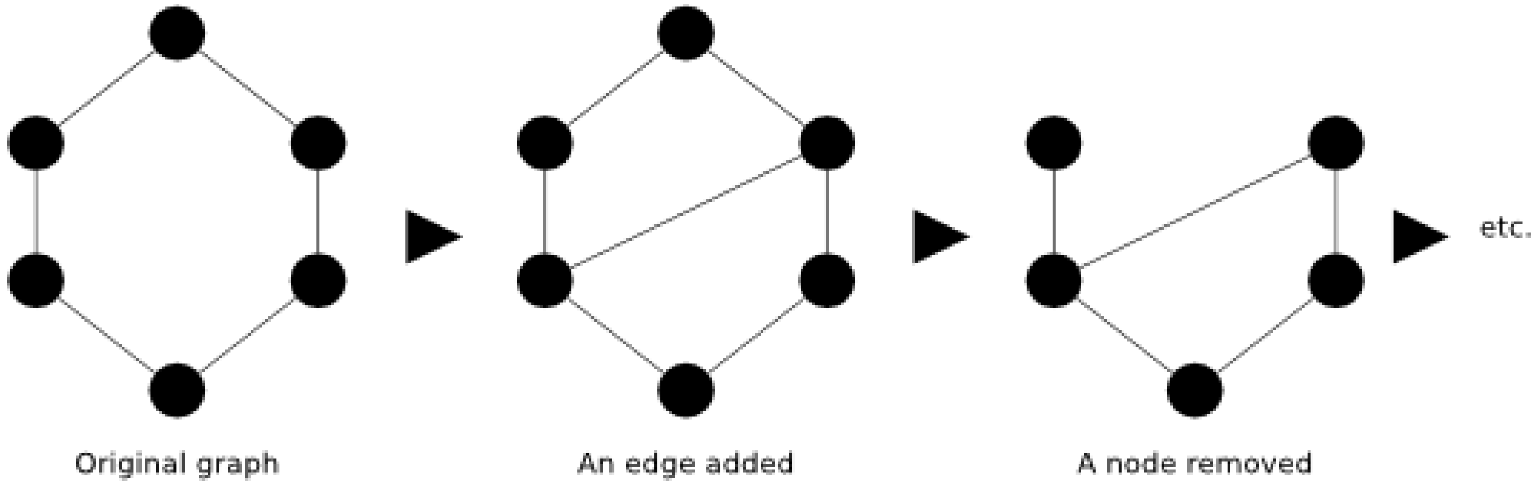


Using compression on static graphs

[1] – WebGraph framework (*2004*)

[2] – k^2-tree data structure (*2014*)

But how to represent dynamic graphs?



The static k^2-tree

Static graphs: k^2-tree is an option

k^2-tree: represents static graphs and binary relations in general

A compressed representation of the data

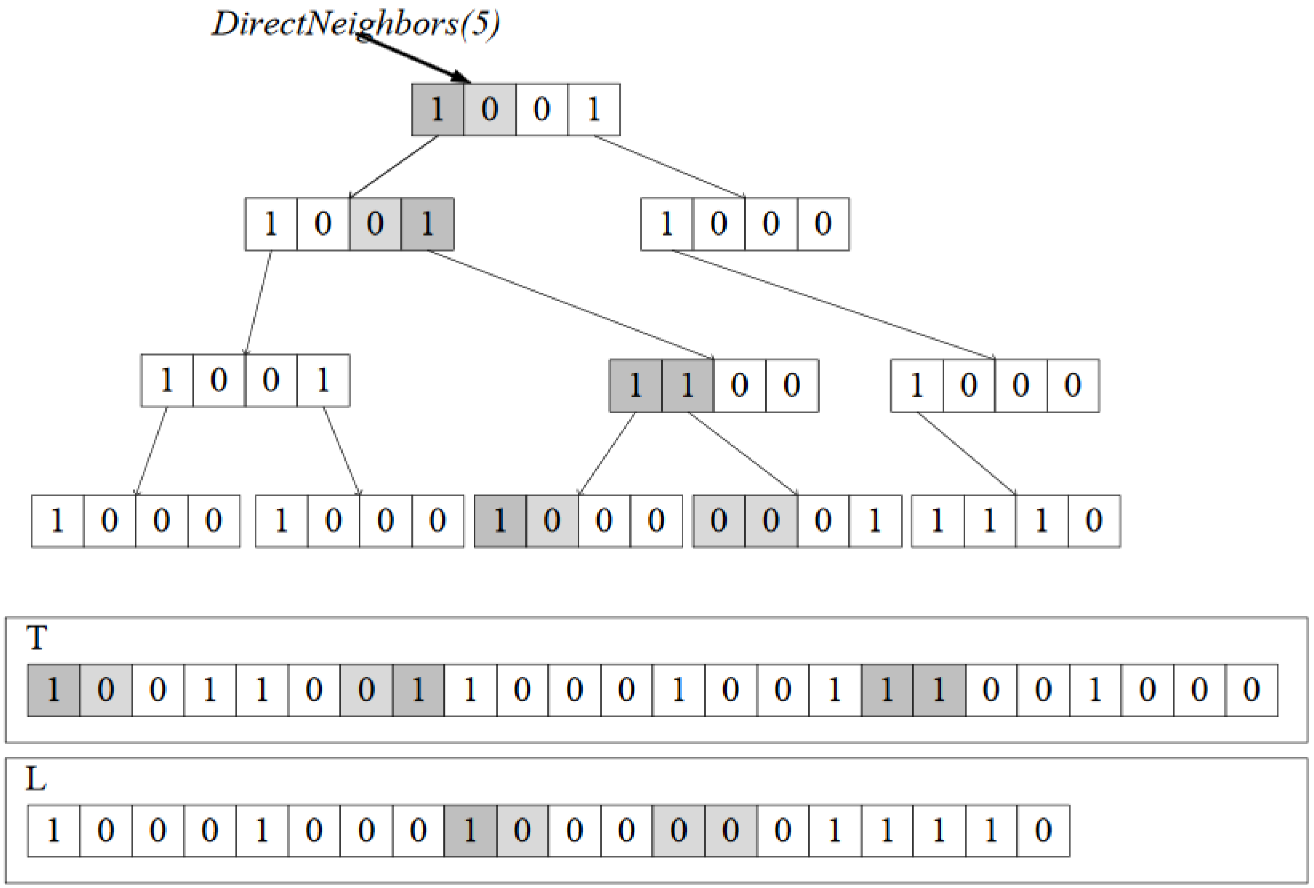
Represents the adjacency matrix of a graph using a **non-balanced k^2-ary tree**

Example uses: web graphs, social networks, RDF datasets…

The static k^2-tree

K=2

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0



Introduction

From static to dynamic k^2-trees

Motivation: dynamic k^2-tree

Static k^2-tree: relies on compact bit vectors

Dynamic k^2-tree: uses compact representations of **dynamic** bit vectors [3] (*2017*)

Problem: bottleneck in compressed dynamic indexing [4, 5]

An alternative k^2-tree implementation

Munro *et al.* [4]: techniques to **dynamize static collections**

- Alternative dynamic k^2-tree implementation
- Edge insertion time almost the same as the average construction time per edge of the static k^2-trees

From static k^2-tree to dynamic graphs

Collection of edge sets $\mathcal{C} = \{E_0, \ldots, Er\}$

Static edge sets E_i with $i > 0$ represented with a static k^2-tree

E_0 represented as dynamic uncompressed adjacency list with hash table for lookups

From static k^2-tree to dynamic graphs

Munro *et al.* [4] – we need to control:

- \# edges m_i in each set E_i
- \# r of such sets

Max. number of edges per set follows a **geometric progression**

$r \leq \frac{2}{\varepsilon}$ for $m \geq 3$

E.g. when $\varepsilon = 1/4, r$ is at most $2/(1/4) = 8$

Furthermore:

- $|E_0|$ represented by m_0 which is at most $m/\log^2 m$
- $|E_i|$ represented by m_i which is at most $m/\log^{2-i\varepsilon} m$

From static k^2-tree to dynamic graphs

Insertions, deletions and queries

Rely on efficient set operations over k^2-trees [6]

For C_1 and C_2 represented as k^2-trees, we can compute:

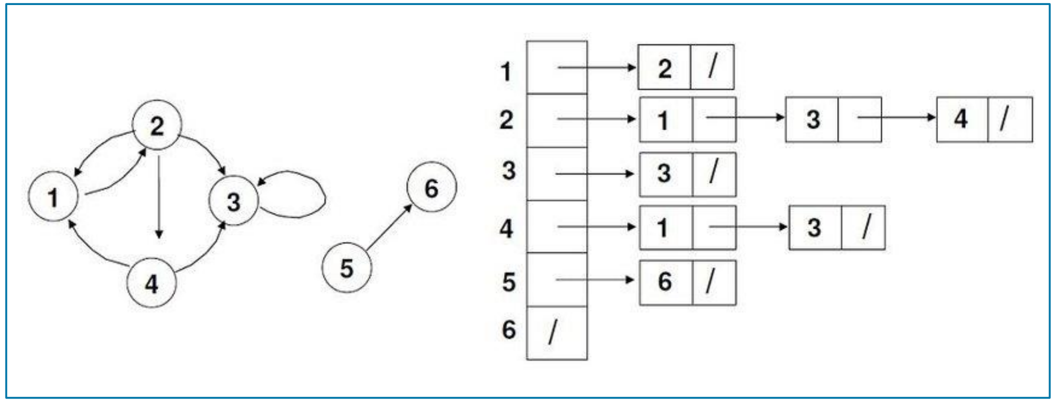
$$C_1 \cup C_2,\ C_1 \cap C_2 \text{ and } C_1 \setminus C_2$$

Linear time on the size $|C_1|$ and $|C_2|$

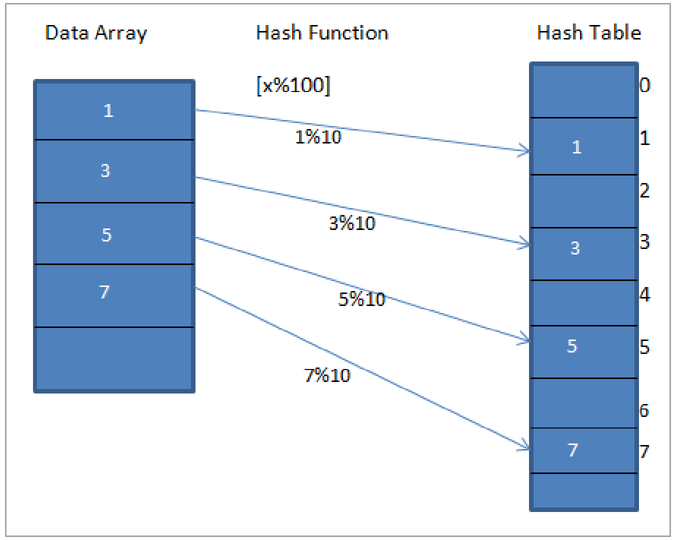
Without decompressing the k^2-trees

From static k^2-tree to dynamic graphs

Insertion of new edge (u, v) when E_0 has space



Uncompressed adjacency list



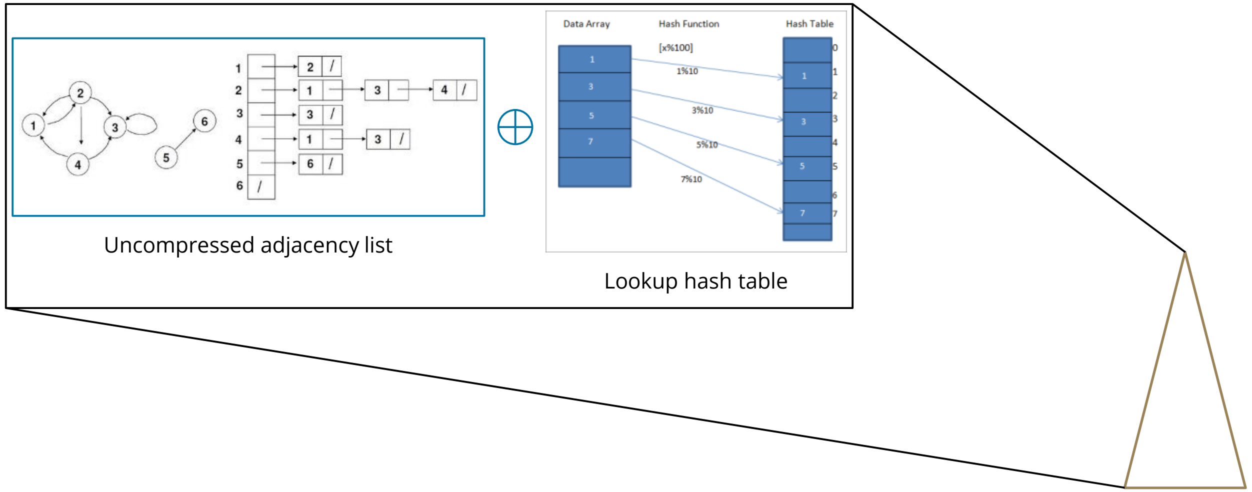
Lookup hash table

E_0 set

Insert edges in E_0 while $|E_0| < m_0 = O(m \log^2 m)$

From static k^2-tree to dynamic graphs

Insertion of new edge (u, v) when E_0 has space



E_0 set

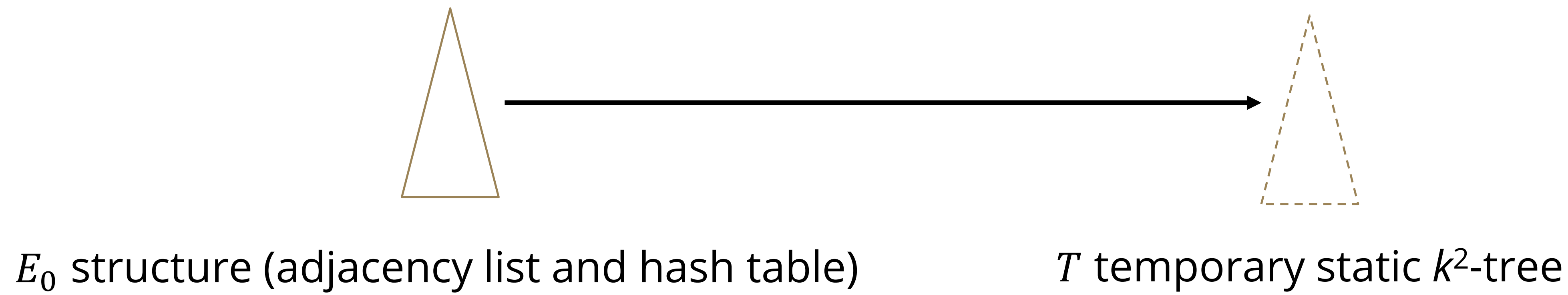
Insert edges in E_0 while $|E_0| < m_0 = O(m \log^2 m)$

From static k^2-tree to dynamic graphs

Insertion of new edge (u, v) when E_0 is full

1 – Create a temporary k^2-tree T with edges of E_0

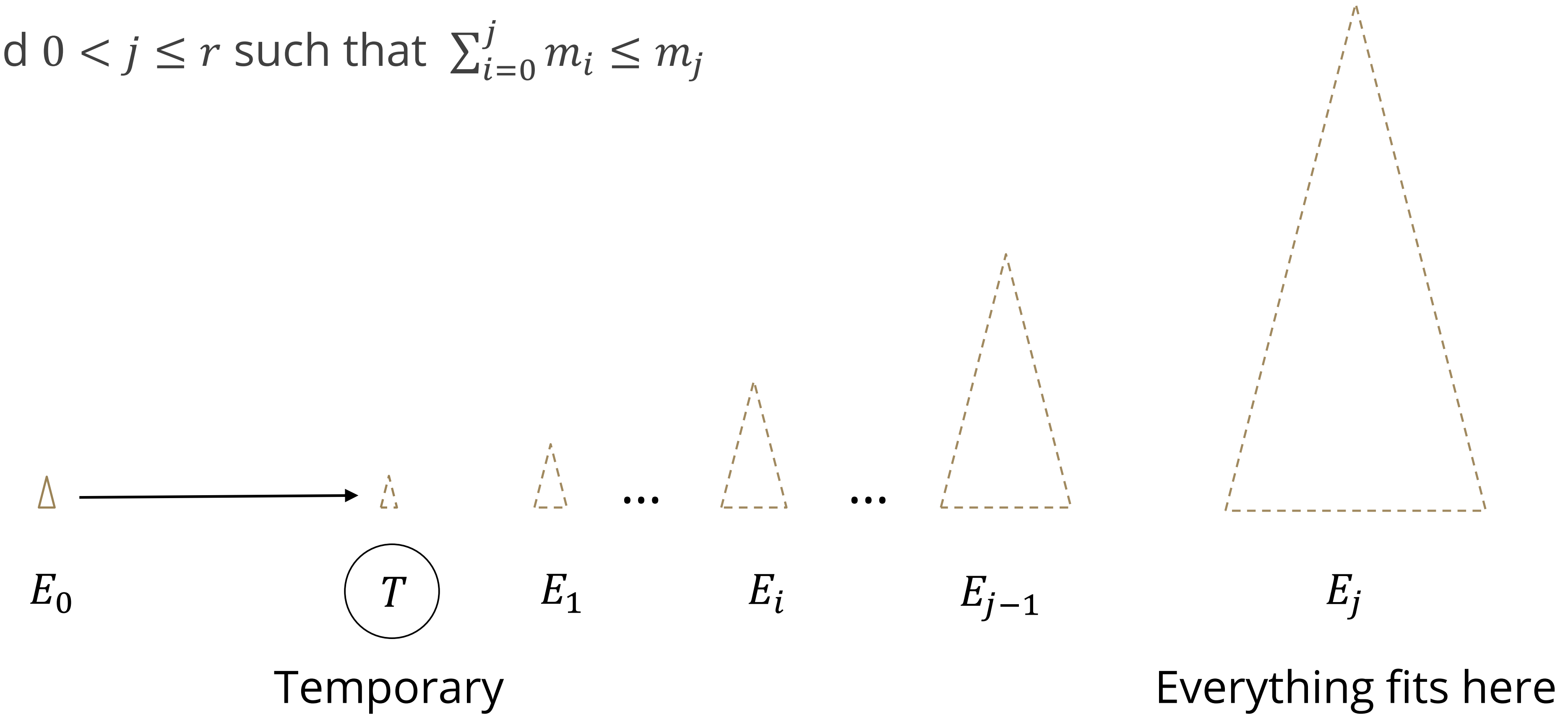
Takes $O(m_0 \log_k n)$ time [2]



From static k^2-tree to dynamic graphs

Insertion of new edge (u, v) when E_0 is full

2 – Find $0 < j \leq r$ such that $\sum_{i=0}^{j} m_i \leq m_j$



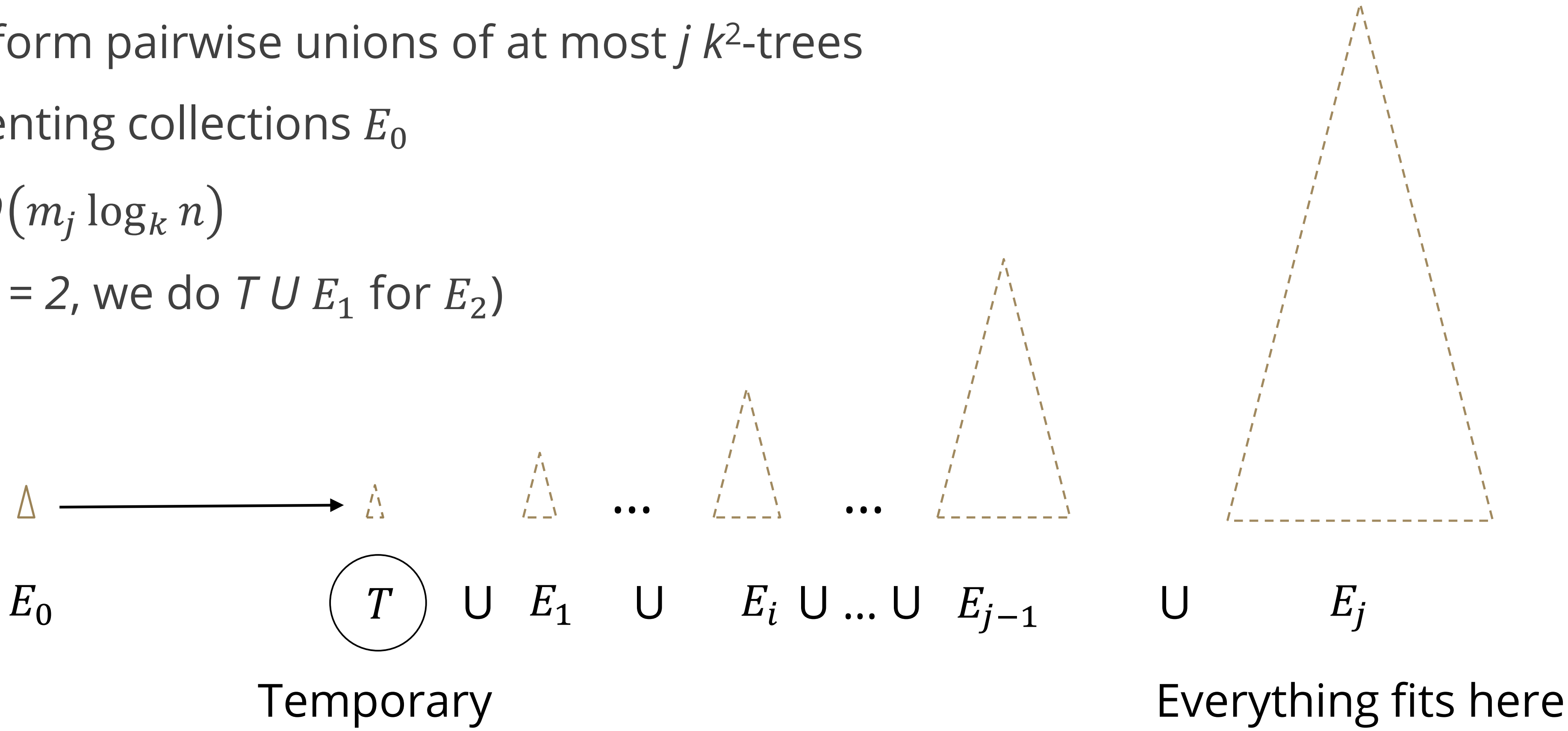
From static k^2-tree to dynamic graphs

Insertion of new edge (u, v) when E_0 is full

3 – Perform pairwise unions of at most *j* k^2-trees representing collections E_0

Time: $O(m_j \log_k n)$

(e.g. if *j* = *2*, we do *T U* E_1 for E_2)



From static k^2-tree to dynamic graphs

Insertion of new edge (u, v) – complexity

- Insertion in E_0 takes constant time (adjacency list plus hash table)

- If E_0 is full, constructing a k^2-tree for it takes $O(m_0 \log_k n)$ [2]
- Pair-wise union of at most *j* k^2-trees representing $E_0 \ldots E_{j-1}$ takes $O(m_j \log_k n)$ time
- Either E_j is new and m has at least doubled, in which case amortized cost per edge insertion is $O(\log_k n)$
- Or E_j already exists and we are adding all edges in collections $E_0 \ldots E_{j-1}$ which are at least $m_{j-1} = m_j / \log^{\varepsilon} m$ edges
- Amortized cost of inserting an edge is therefore $O(\log_k n \log^{\varepsilon} m \,(1/\varepsilon))$

From static k^2-tree to dynamic graphs

Deletion of edge (u, v)

If (u, v) is in E_0 just remove it

Else find $0 < j \leq r$ such that $(u, v) \in E_j$

If found

Set bit to 0 in the E_j k^2-tree

Update number of deleted edges m'

If $m' > m / \log\log m$, rebuild $\mathcal{C}$ – costs $O(m \log_k n)$

From static k^2-tree to dynamic graphs

Deletion of edge (u, v) *- complexity*

- Deletion in E_0 takes constant expected time
- Checking and deleting in our collection $\mathcal{C}$ takes $O((\log_k n)/\varepsilon)$ time
  - Checking if an edge exists in a given k^2-tree takes $O(\log_k n)$ [2]
  - Might have to look in each collection E_i with $0 < i \leq r = \lceil 2/\varepsilon \rceil$
- Full rebuild after $m/\log\log m$ edge deletions costs $O(m \log_k n)$
  - Amortized cost per deleted edge of $O(\log_k n \log\log m)$
  - Overall amortized edge deletion cost is $O((\log_k n)/\varepsilon + \log_k n \log\log m)$

From static k^2-tree to dynamic graphs

Querying of edge (u, v)

Works like in the static k^2-tree implementation

But need to query all sets in the collection

This increases the cost by a factor of $O(1/\varepsilon)$ vs static k^2-tree

From static k^2-tree to dynamic graphs

Comparison with other constructions (time)

Our implementation

Operations	k2tree[2]	dk2tree[3]	sdk2tree	k2trie[7]
Insert time	$O(\log_k n)$**	$O(\log_k n \log n)$	$O(\log_k n \log^{\varepsilon} m)$*	$O(\log_k n)$*
Delete time	N/A	$O(\log_k n \log n)$	$O((\log_k n)\,(1/\varepsilon + \log\log m))$*	N/A
Query time	$O(\log_k n)$	$O(\log_k n)$	$O((1/\varepsilon)\log_k n)$	$O(\log_k n)$
List time	$O(\sqrt{m})$**	$O(\sqrt{m})$**	$O(\sqrt{m})$**	N/A

* denotes amortized time

** denotes average time

From static k^2-tree to dynamic graphs

Comparison with other constructions (space)

Implementations	Space (bits)
k2tree[2]	$k^2 m(\log_{k^2}(n^2/m) + O(1))$
dk2tree[3]	$k^2 m(\log_{k^2}(n^2/m) + O(1))$
sdk2tree	$k^2 m(\log_k(n^2/m) + 2\log\log n) + O(k^2/\varepsilon) + o(m)$
k2trie[7]	$O(m\log(n^2/m) + m\log k)$

Experimental Analysis

Setup, methodology, datasets, results

Setup

Implementations written in C

- Single-threaded
- Compilation: gcc 6.3.0 2017-05-16 with -03 optimization

SMP machine

- 256GB RAM
- 4 Intel(R) Xeon(R) CPU E7-4830 @ 2.13GHz
  - Cache: L1 – 512KB, L2 – 2MB, L3 – 24MB
  - 8 cores, 64 threads total

Methodology

Dynamic structures `dk2tree`, `sdk2tree` and `k2trie{1,2}` initialized empty

`k2trie{1,2}` – different parameters for speed/space tradeoffs

- `k2trie1: space efficiency`
- `k2trie2: operation speed`

Addition: add all edges

Deletion: add all edges and then remove 50%

Listings: add all edges then query 50% of the vertices

Queried/removed edges and listed vertices were sampled offline to allow reproducibility

Peak resident memory: `GNU time`

Datasets

Dataset	$\|V\|$ (M)	$\|E\|$ (M)	k2tree (bit/edge)	dk2tree (bit/edge)	sdk2tree (bit/edge)	k2trie1 (bit/edge)	k2trie2 (bit/edge)
dm50K	0.05	1.11	21.10	23.64	21.26	43.16	298.99
dm100K	0.10	2.59	22.66	25.27	22.76	47.31	257.61
dm500K	0.50	11.98	27.87	30.85	27.97	57.92	187.91
dm1M	1.00	27.42	29.48	32.63	29.49	58.78	132.92
uk-2007-05	0.10	3.05	2.98	3.39	3.16	5.62	11.11
in-2004	1.38	16.92	2.99	3.40	3.14	3.90	6.97
uk-2014-host	4.77	50.83	9.47	10.55	9.58	13.07	21.88
indochina-2004	7.42	194.11	2.46	2.79	2.59	2.88	4.91
eu-2015-host	11.26	386.92	5.61	6.26	5.71	7.02	11.64

Top: generated with duplication model

Bottom: obtained from Laboratory of Web Algorithmics [8, 9]

Datasets

Synthetic datasets generated with partial duplication model [10]

Captures abstraction of real-world datasets in a simple way

But global statistical properties of biological networks are well captured [11]

Fig. 1: average time for adding edges

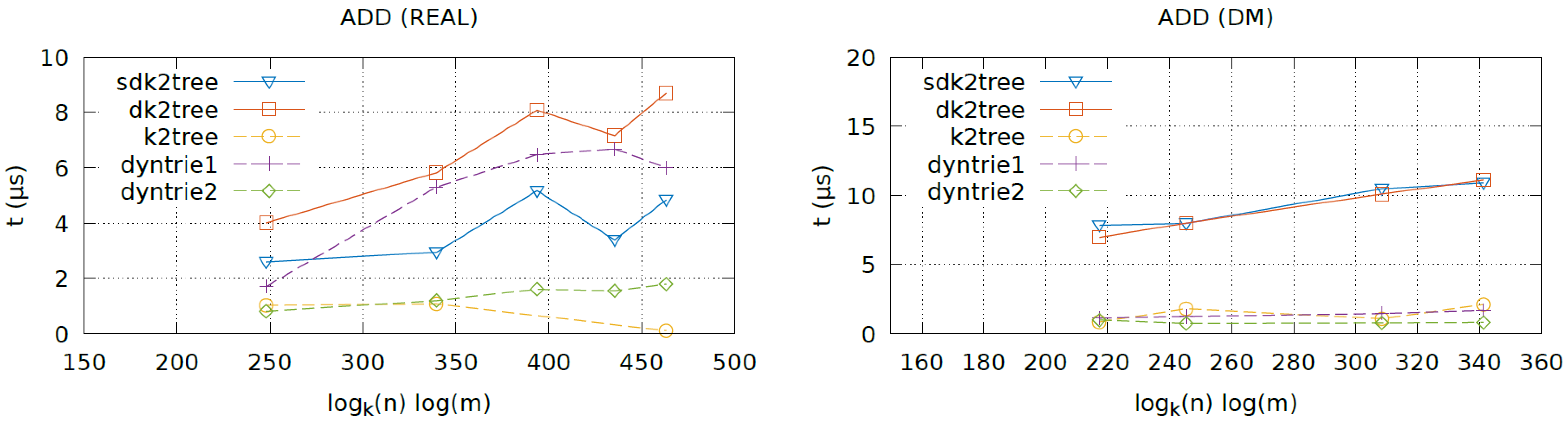
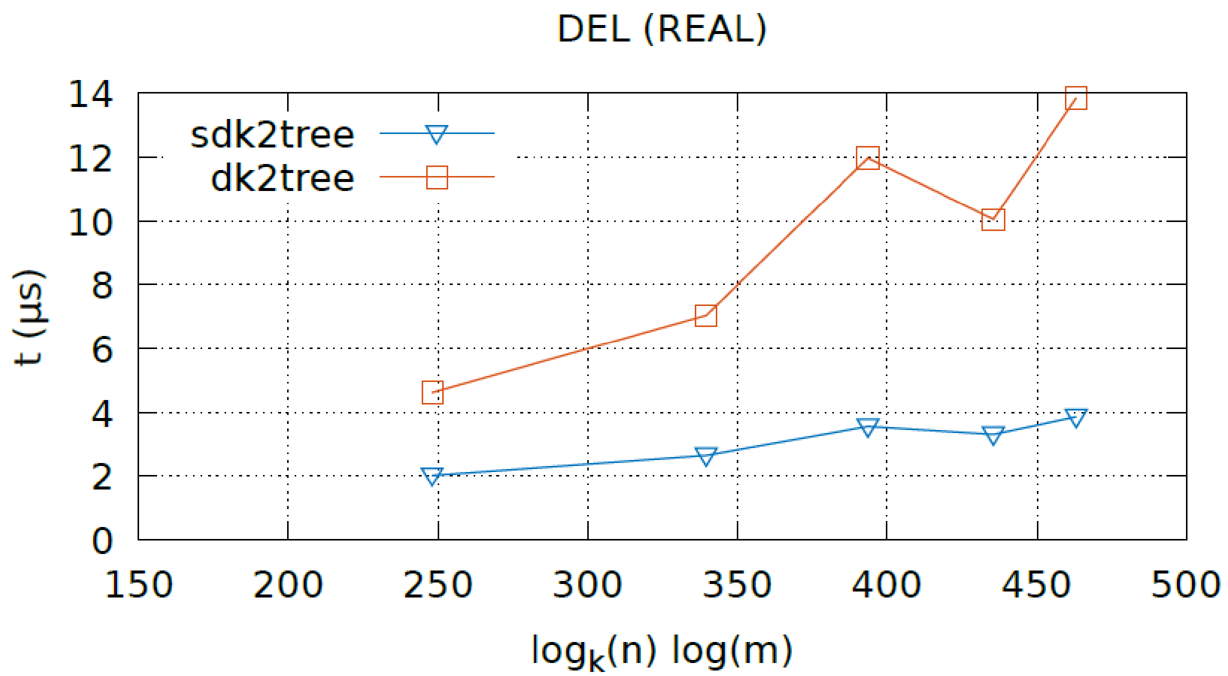


Fig. 2: average time for deleting edges



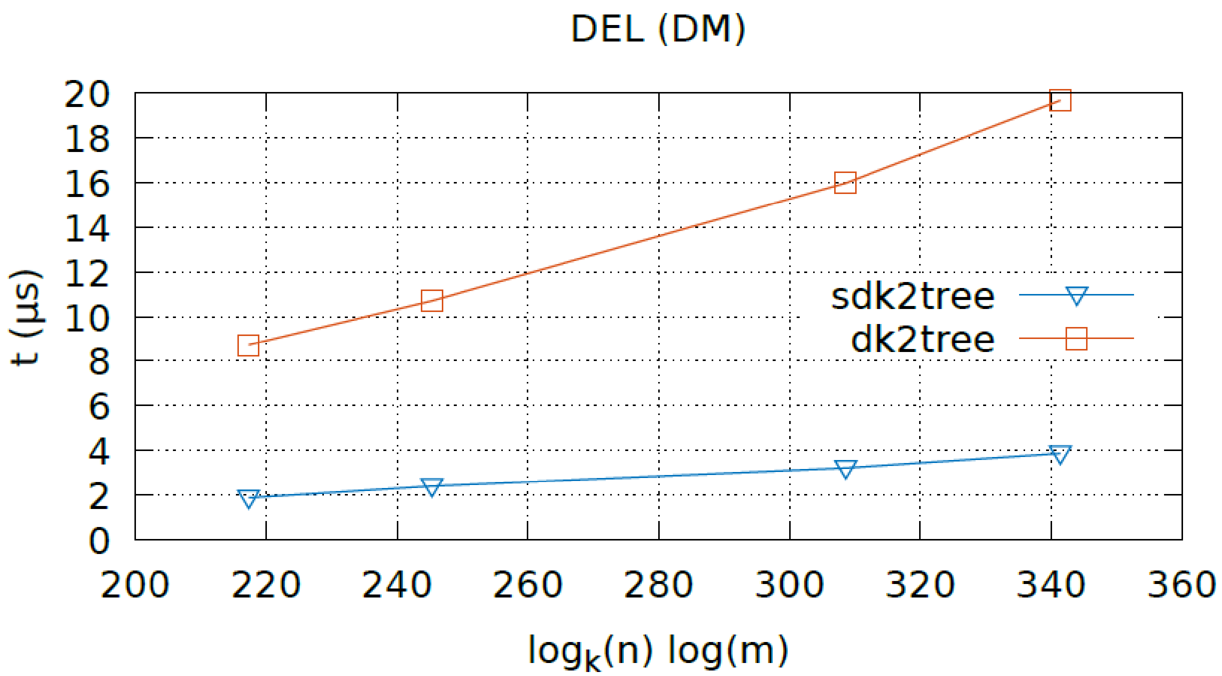


Fig. 3: average time for listing neighbors

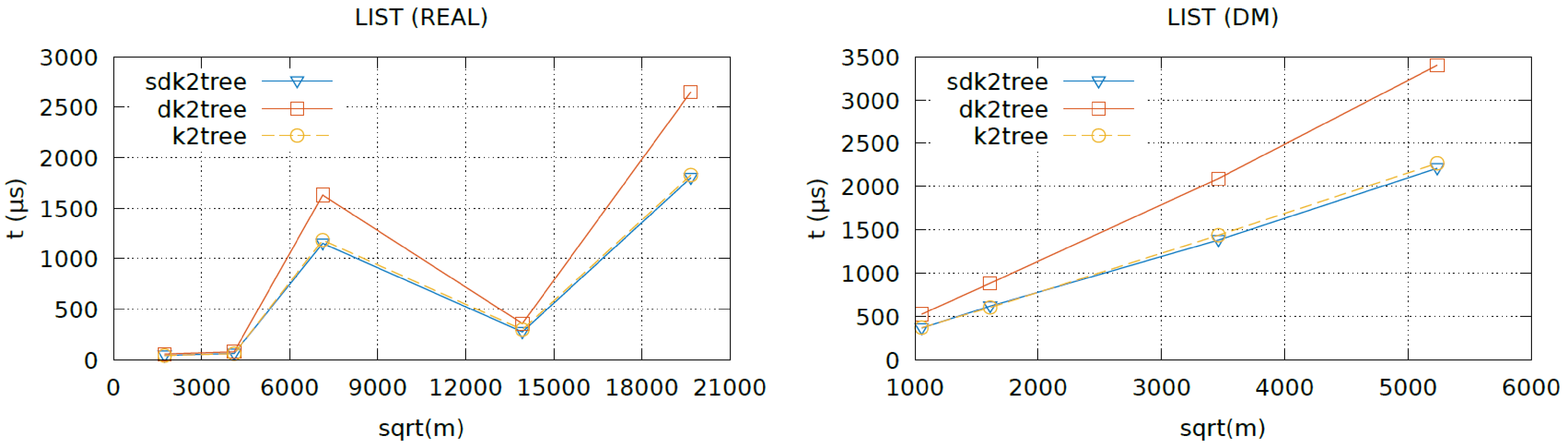


Fig. 4: average time for checking edges

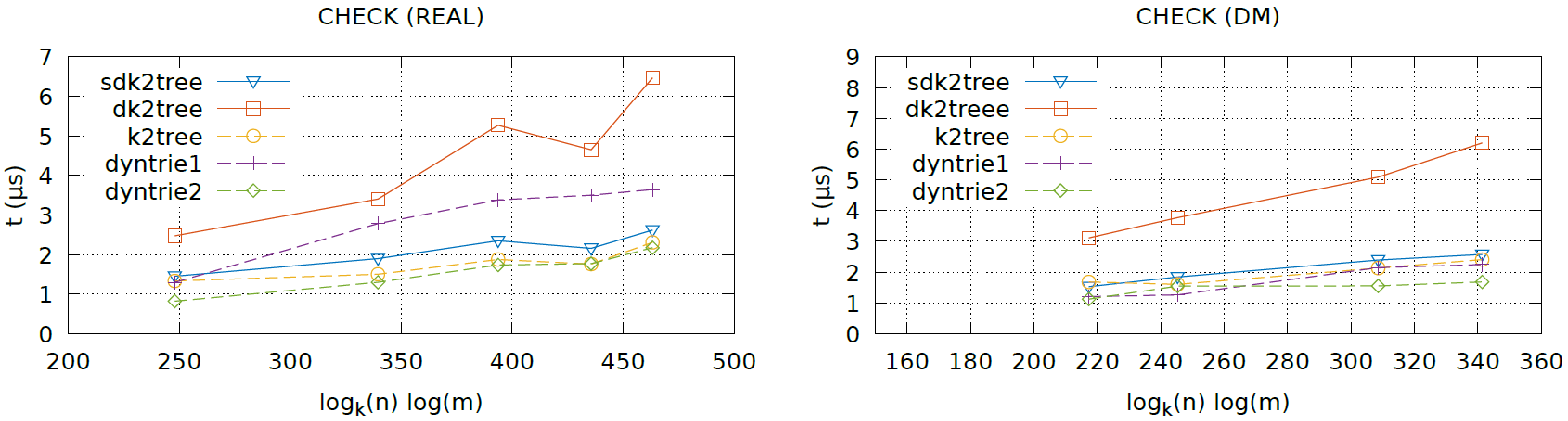


Fig. 5: max resident memory while adding edges

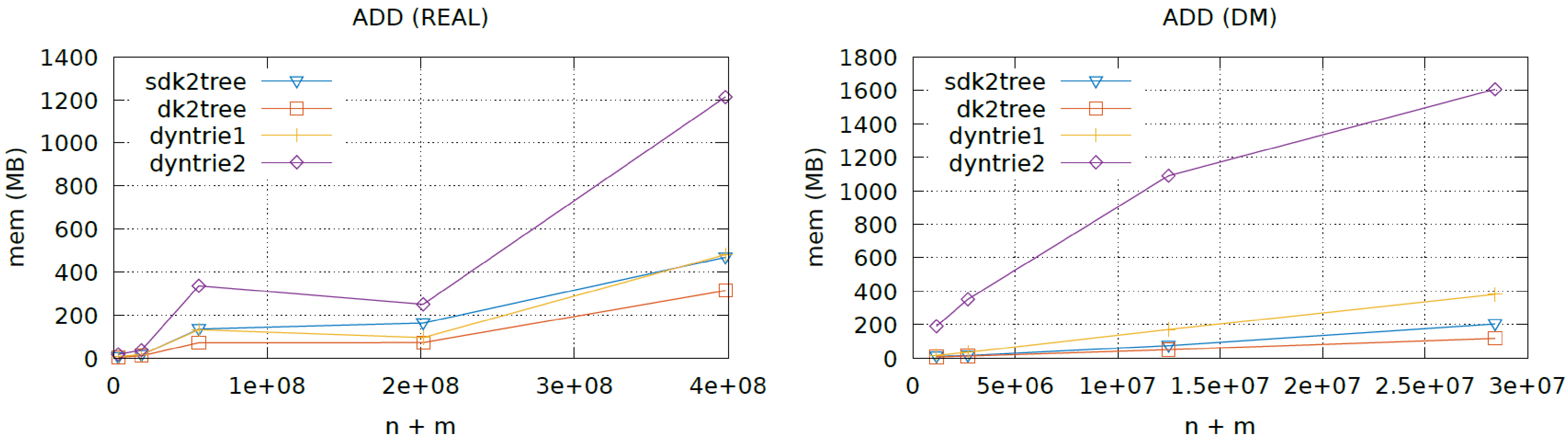


Fig. 6: max resident memory while deleting edges

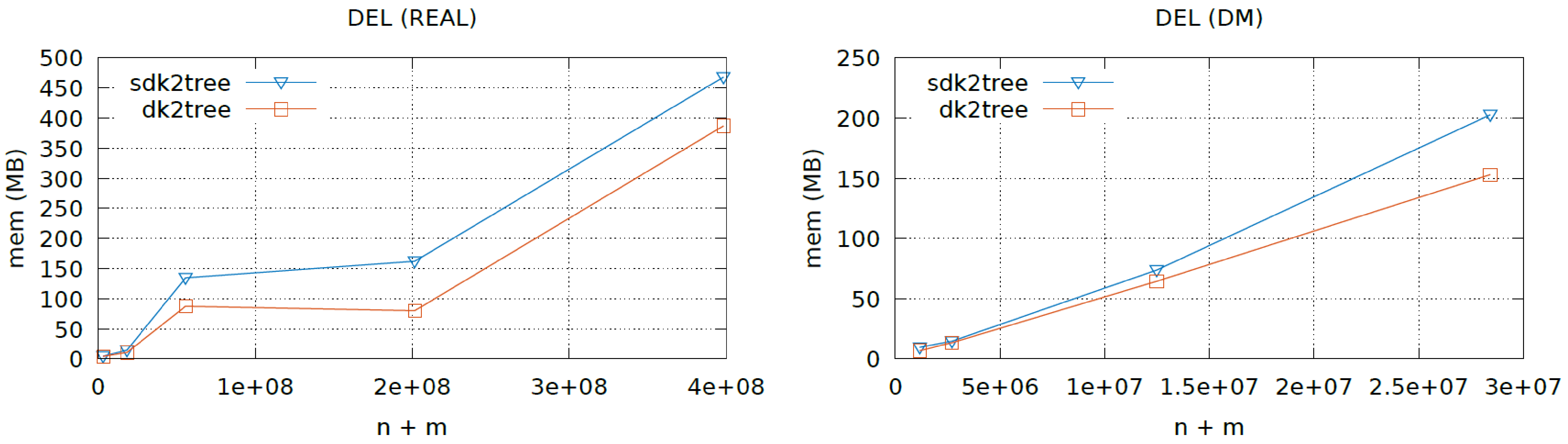
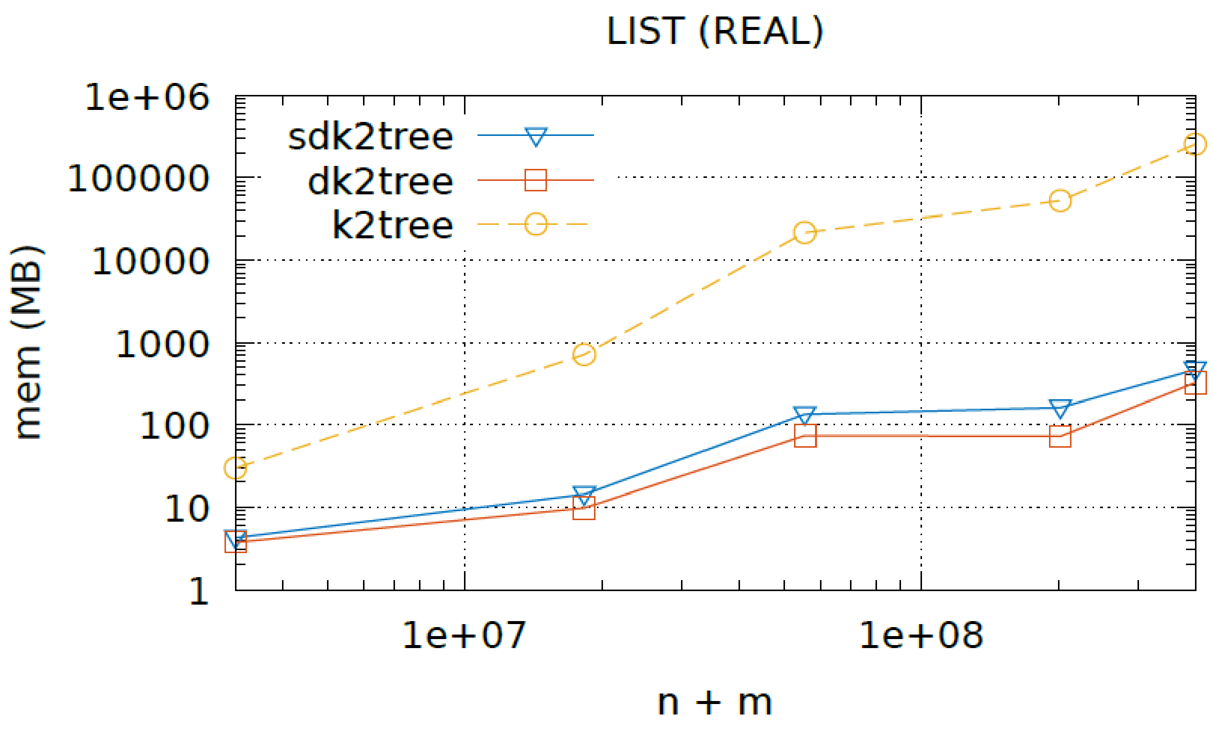


Fig. 7: max resident memory while listing neighbors



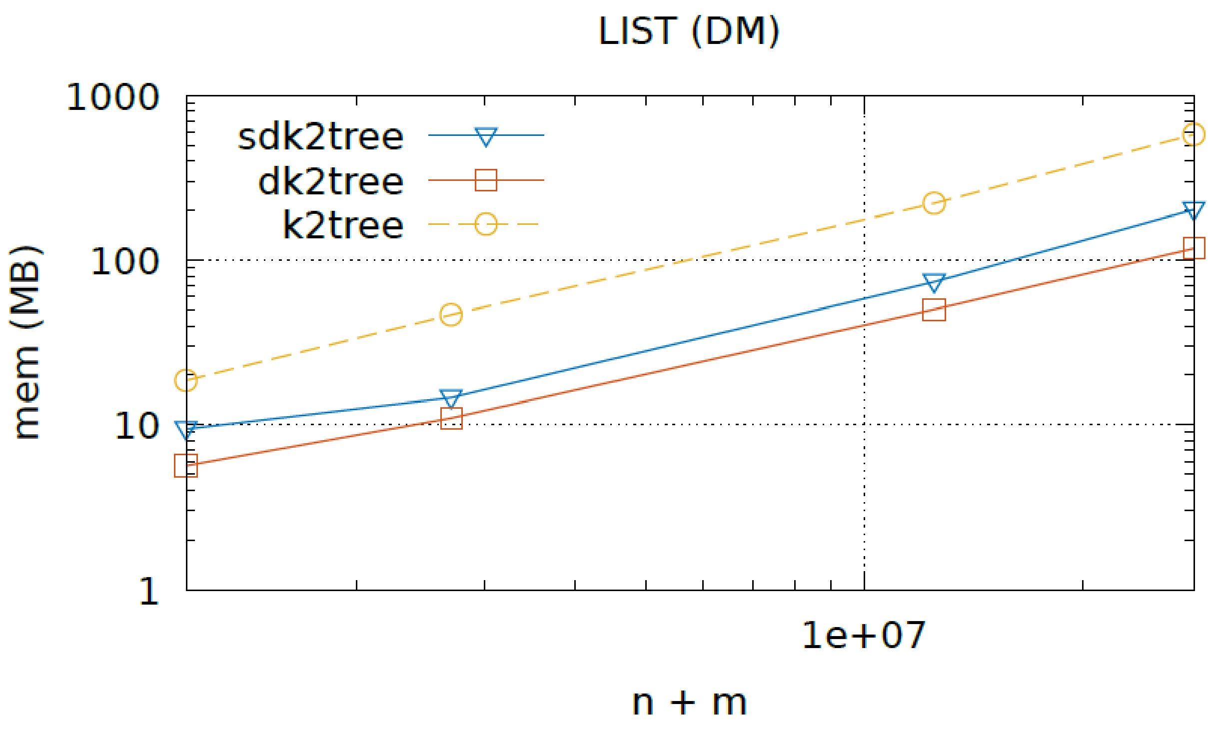
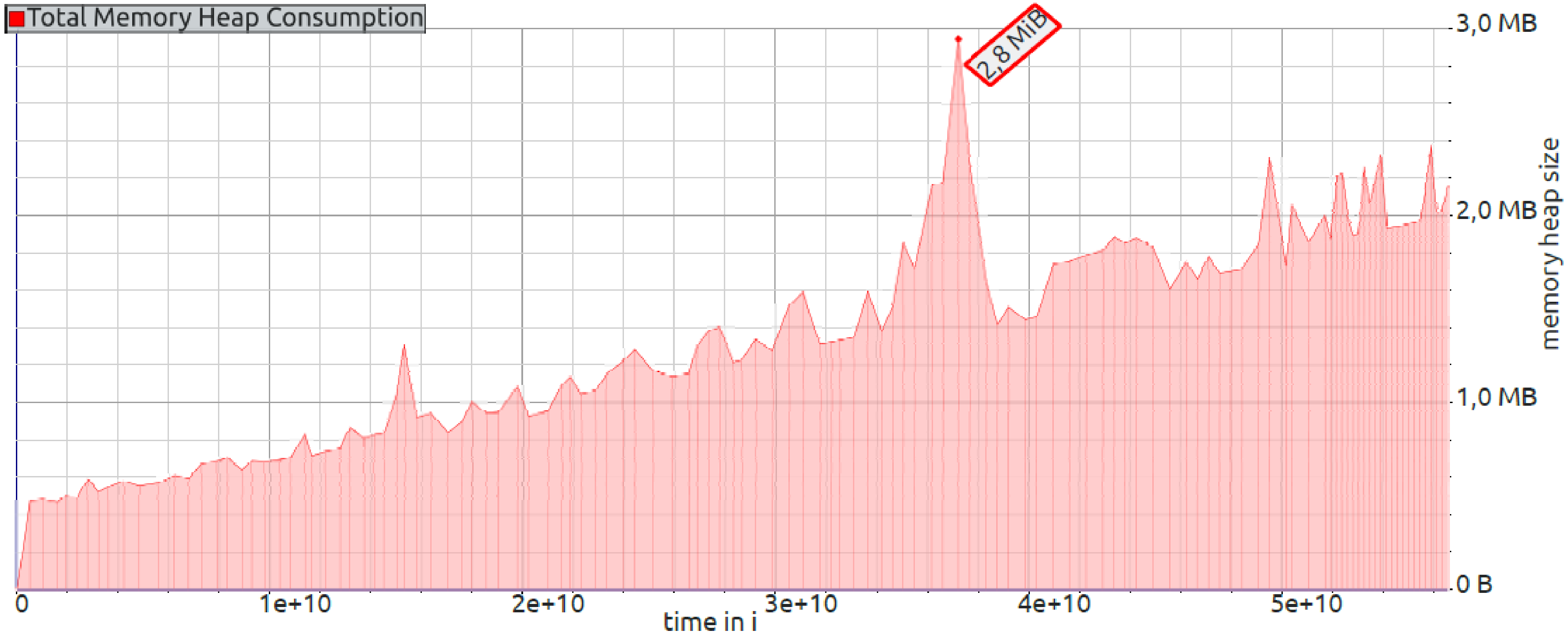


Fig. 8: `valgrind` heap allocation profile for `uk-2007-05`.

Label `time in i` in the x axis is the *#instructions* executed

`valgrind --tool=massif –max-snapshots=200 –detailed-freq=5`

Sets: $\{E_1, \ldots, E_8\}$, #unions: $508, 127, 63, 32, 17, 8, 4, 1$



Conclusion

Major highlights

Take-home

`sdk2tree`: semi-dynamic data structure (based on a collection of static k^2-trees)

Additions and removals with competitive performance

Faster times than `dk2tree` [3] dynamic bit vector version

On par with `k2trie` [7] regarding additions and queries

For the future:

- Refine data structure, potentially as a library