Adv. Data Structures: van Emde Boas Trees

Anders Ingemann (20052979)
Peter E. Hvidgaard (20062546)

November 21, 2011

Project

yadyyady om projektet praktiske ting som hvor kildekode ligger hvilken test maskine vi bruger hvordan man compiler og kør stødte vi ind i nogen problemer andet

van Emde Boas Trees

The van Emde Boas Tree data structure is recursive, this means we have to decide when to end the recursion. A natural starting point is to let the recursion end with leafs consisting of the 2 elements, *min* and *max*, but since a tree with 3 elements will have a *TOP* and *BOTTOM* with a single element, that must also be supported. However, as stated in the note from G. Frandsen, and at the lectures, there is a point where you will gain better performance by reverting to an array and a liniar scan. Our implementation allows for this *threshold* to be set upon creation of the tree. Quick testing showed that a threshold between 32-256 gave roughly the same performance. Higher or lower affected the query-time negatively.

Supported operations

Our implementation support the following queries on a vEB tree:

- $uint32\_t$ **veb_insert**$(uint32\_tindex, void * data, vebtree * tree)$
- $void$ **veb_delete**$(uint32\_tindex, vebtree * tree)$
- $int32\_t$ **veb_findsucc**$(uint32\_tindex, void * data, vebtree * tree)$
- $int32\_t$ **veb_findpred**$(uint32\_tindex, void * data, vebtree * tree)$

delete_min and **find_min** can both be implementeted with the above operations, by first calling **veb_findsucc** on 0, and if you want to delete it, call **veb_delete** on the returned index.

Handling the leafs

Our leafs are a simple array of elements and as such will not be explained in more detail than this part. The operations to **insert** and **delete** are a matter of inserting and deleting in the correct place in the array, this is done in constant time. The operations **find_pred** and **find_succ** are a matter of scanning the array from a certain posistion to find the first, if any, element that is before or after. The *min* and *max* is kept in a seperat pointer, so

finding and accessing them is constant time. Since the arrays have a fixed length, independent from the size of the universe, **find_pred**, **find_succ** are also constant time.

Handling the recursive nodes

The trees are build as described on the slides from the lectures - and **insert**, **delete** and **find_succ** are all implemented as described on the slides as well. The operation **find_pred**, is mostly identical to **find_succ**, but obviously look for the first element preceeding the index. It's basicly a mirror of **find_succ**.

Futher improvements

Out leafs are an ordinary "nodes" with an array instead of continuing the search. One possibility could be to use a leaf size of 32 or 64, depending og the machine, and represent the elements as a bit array. Manipulating this will be extremly efficint, and then keep pointers to the actual data somewhere else.

Comparison with priority queues

The vEB tree can be used as a priority queue if we build some extra logic on top of it. Firstly a priority queue can contain several elements with the same priority, a search tree does not allow this. To get around this limitation we've build a new priority queue that can use any search tree as the underlying data structure. Ignoring intialization (this is just initializing the underlying search tree), we focus on the actual usage. Our priority queue support the following operations:

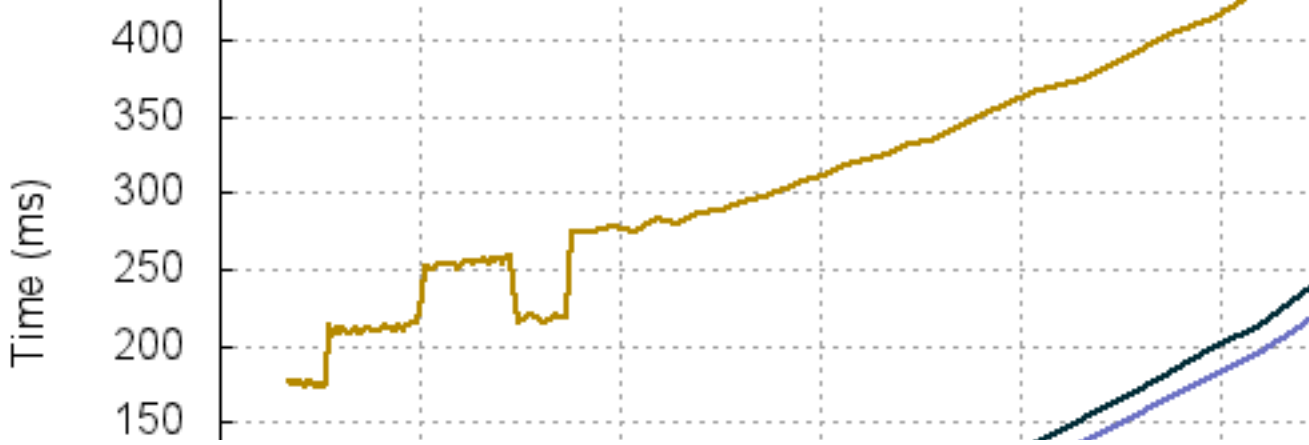
- *void* **veb_pq_insert**$(veb\_pq\_node * n, vebtree * tree)$
- *void* **veb_pq_delete**$(vebtree * tree, veb\_pq\_node * node)$
- $veb\_pq\_node *$ **veb_pq_deletemin**$(vebtree * tree)$
- *void* **veb_pq_decrease_key**$(vebtree*tree, veb\_pq\_node*node, uint32\_tdelta)$

A **veb_pq_data** is just a simple structure that contain a pointer to the first **veb_pq_node** in a linked list of nodes, and a counter, n that is the number of nodes in the linked list. A **veb_pq_node** contains a pointer to the next and previous nodes, if any, and a pointer to the **veb_pq_data** holding the node; in particular, this last pointer allows us to check if a node is in the search tree or not, but seeing if the pointer is $NULL$. It also have an unsigned integer field that is the priority. Any auxilary data should also be included in this node - we just have a node number to indicate which node in the graph for our dijkstras algorithm it represent.

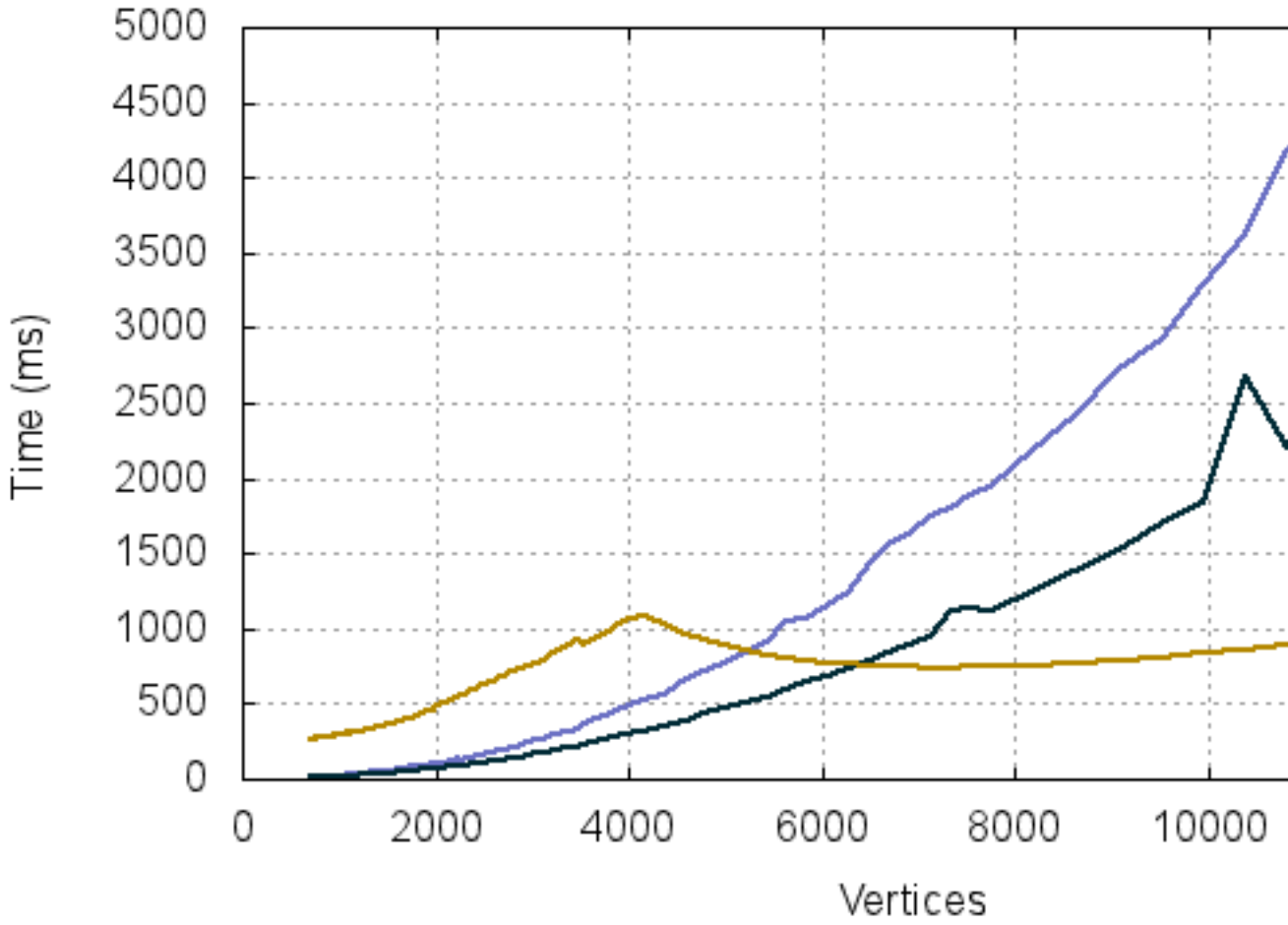
veb_pq_insert

kort om BinHeap og FibHeap

We have chosen to benchmark the van Emde Boas tree with the two graph generation algorithms we implemented in the previous assignment. Those include a random graph generator and a generator which should cause a high number of decrease_key calls when using dijkstra on the graph to solve the single source shortest path problem. The random graph generator generates a graph, where each vertice has a 15% chance of being connected to another vertice with the weight being between 1 and a chosen maximum.

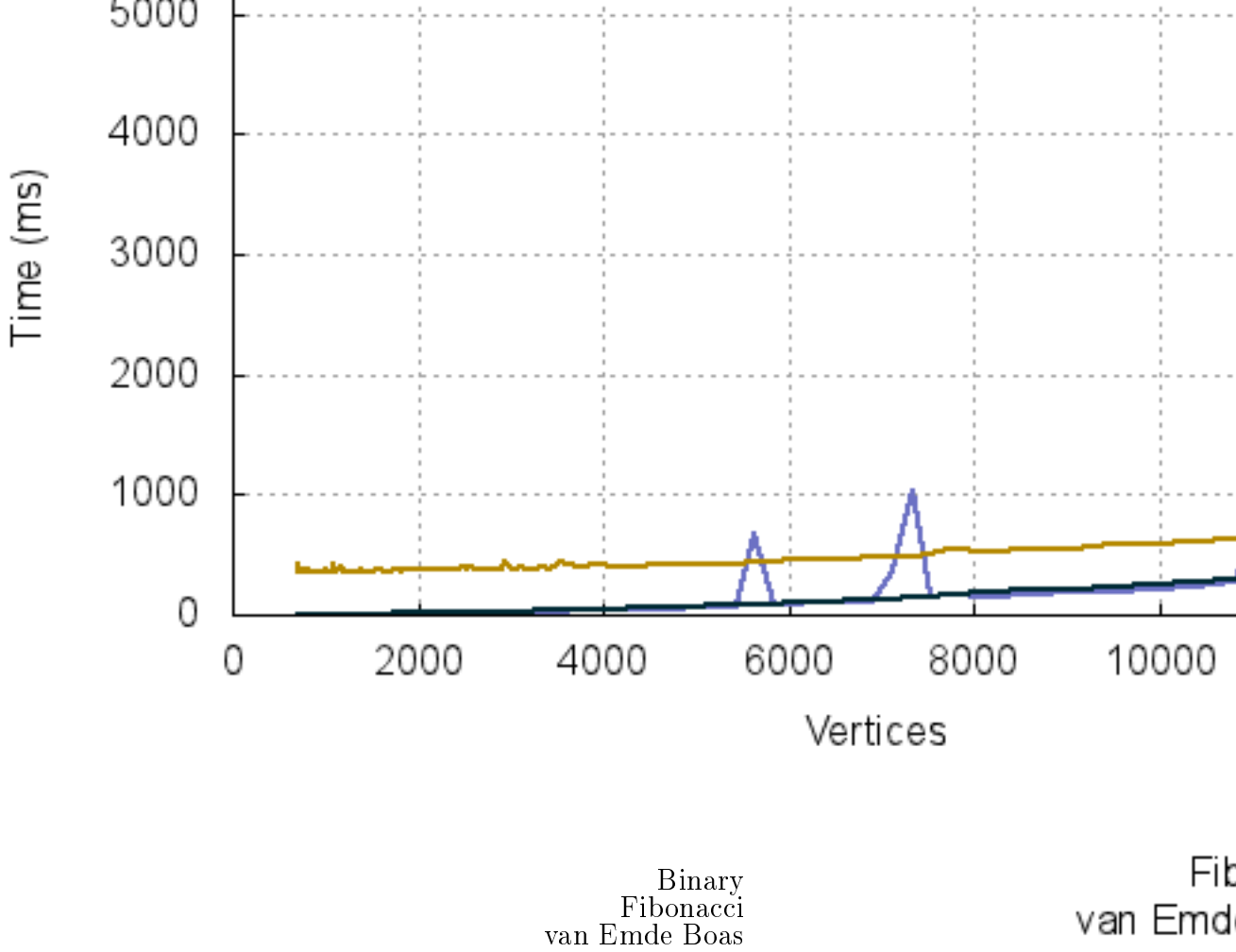


Decrease Key maximized v2 graph averag

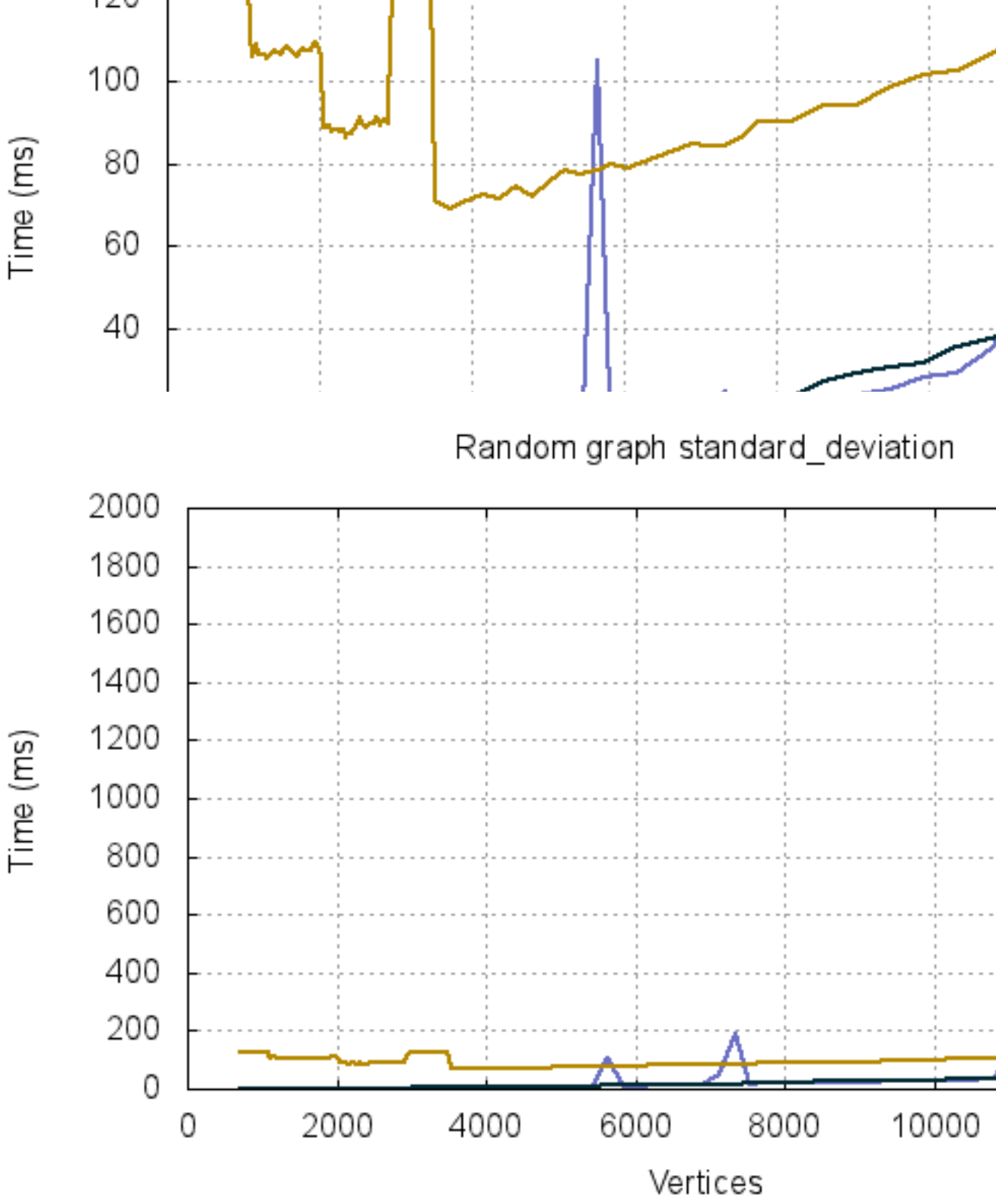


Binary
Fibonacci
van Emde Boas

Fib
van Emd



We measured the running time using two different methods. One using normal absolute time measurement, the other one counting clock cycles. Curios to see which method had the lowest standard deviation we were surprised to discover that it increases using both methods quite rapidly as the graph sizes grow bigger.



We have yet to explain why this is happening.

hvilke test cases har vi og hvorfor? test og plots konklutioner på testne

Comparison with Red-Black Trees

Kort om red-black trees hvilke test cases har vi og hvorfor?

In order to compare Red-Black trees to van Emde Boas trees we constructed a simple sorting benchmark. To do that we implemented a list generator, whose product should be sorted by either algorithm. The generator generates a list of random numbers from 0 to a given maximum and terminates once a given size is reached.

test og plots konklutioner på testne

Appendix

6

Bibliography

[1] Thomas H. Cormen et. al., *Introduction to algorithms*, 2nd Edition.