Applied Algorithms - Exam Paper

By
Jonas Hartmann (haja@itu.dk)
Gabriel Agger Nielsen (gaag@itu.dk)
Lukas Christian Nielsen (luni@itu.dk)

Applied Algorithms, KSAPALG1KU, F2019.
IT University of Copenhagen.

December 19, 2019

Contents

1 Part 1 **3**
1.1 Exercise 1 . . . 3
1.1.1 Memory layouts . . . 3
1.1.2 OtherArray implementation . . . 3
1.2 Exercise 2 . . . 3
1.2.1 Input generation . . . 3
1.2.2 Choices . . . 3
1.2.3 Finding optimal value for **p** . . . 4
1.2.4 Results . . . 4

2 Part 2 **4**
2.1 Exercise 3 . . . 4
2.2 Exercise 4 . . . 5
2.2.1 Scenario tests . . . 5
2.2.2 Random tests . . . 5
2.3 Exercise 5 . . . 5
2.3.1 Results . . . 6

1 Part 1

1.1 Exercise 1

1.1.1 Memory layouts

Figures 4 and 6 in [Brodal and Moruz, 2006] plots the running time against the balance parameter α as well as the number of level-1 cache faults against α for the simple (non-blocked) and blocked memory layouts of their implementations of SBST. Their results showed that for the simple memory layouts, `Depth First Search` (DFS) performed best, both in terms of running time and cache faults. For the blocked layouts, skewed van Emde Boas implementation was the fastest.

1.1.2 OtherArray implementation

The `OtherArray` is implemented like a `pqDFS` blocked memory layout. It takes a `SearchTree` that has precomputed the weights of every node contained in a given `bitvector`. The p heaviest nodes are then stored first followed by the subtrees of their children, both are stored in ascending order of their weights. p should preferably be the size of a memory block in the given hardware to minimize the number of I/Os [Brodal and Moruz, 2006].

1.2 Exercise 2

1.2.1 Input generation

Our input generation is controlled by the `Generator` class under `part1` in the accompanying source code. For the values in our set, we simply generate n random integers and place these into an integer array. When generating the sets we ignore how they are sorted as that should not be relevant to the queries; n random integers are put into the set.

The queries consists of values between the min and max of the set, where min and max will **always** be a part of the queries. This is because any values outside of this range will be $O(1)$ for each of our implementations, while either min or max will be our worst case depending on α.

Experimental design	
	Description
Hypothesis	We expect OtherArray to have the fastest average running time per query of the three implementations.
Performance indicators	Average running time per query
Factors	Seed s, Skewness α
Levels	$s = \{7;\ 943;\ 57,438\}$, $\alpha = 0, 05...0.95$ by 0.05
Trials	200 per design point
Design points	$s \cdot \alpha$
Outputs	Average running time per query when given a particular α on a constant size of data and queries.

1.2.2 Choices

Our environmental parameters are Ubuntu 18.04.3 LTS on a Windows 10 $x86\_64$ machine. The machine has an i5-9600k CPU 3.696 GhZ, with a L1 cache of 384 KB and 3,305 GB RAM.

The performance indicator in this experiment is chosen to be the average running time per query, in order to showcase the effect of the memory cache and how the least comparisons does not necessarily mean that it has the fastest average running time.

In this experiment n, q, p are all **fixed parameters**. The reason for choosing $n = 250,000$ is to ensure that the set size will be outside of our level 1 cache, while still having a reasonable benchmarking time. $q = 1,000,000$ is to ensure more reliable results for the average run time per query.

The factor of the **algorithmic parameter** α is set at the given levels to enable realization of the optimal skewness for each binary search tree implementation. The instance parameter s factors at the different levels to minimize possibility of badly generated sets and queries.

The design points are set to $s \cdot \alpha$ to compare the average running query time at each α on different seeds. Each query is repeated 200 times to provide more robust results, while a large number would have been preferred 200 * 1,000,000 queries was chosen because of a time limitation.

1.2.3 Finding optimal value for p

We ran an experiment in order to find the optimal value for the block size p for our `OtherArray` implementation. In this initial pilot study we found the optimal value for the size of our blocks (p) to be $p = 1$ where $n = 25000$ and $q = 1,000,000$. This is essentially the same as a DFS. This is most likely due to us running with a size of n that allowed everything to be stored in the 1st level cache. Since the 1st level cache is of size 386 Kb for our hardware, we would have to use an $n > 96000$. Due to a limitation of time we have been unable to perform the experiment again on this value.

1.2.4 Results

The results from the experiment in Figure 1 shows the average running time per query against the balance parameter α for each of our three implementations. Based on these results our hypothesis seems to be correct. The results from the experiment shows that, as expected, the `OtherArray` implementation was faster compared to the others, at all values of α. Our findings follows those made by [Brodal and Moruz, 2006].

A note in regards to our result is that we use a $p = 1$ which essentially means that our `OtherArray` runs as a depth first search. Our argument for doing so follows the findings from our pilot study, as described in the previous section. We expect, that were we to run the experiment with a sufficiently large value for n we would see a more significant improvement in the performance of `OtherArray` for blocked layouts.

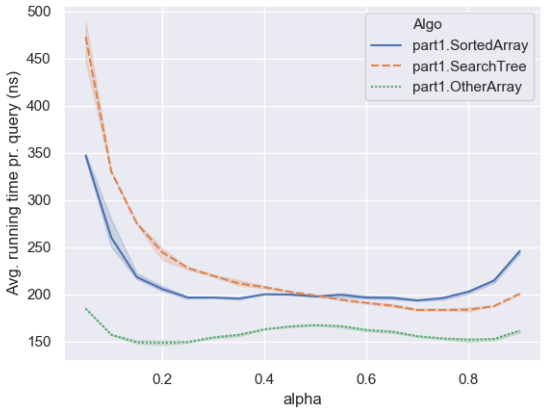

Figure 1 - Avg. running times of SBST

2 Part 2

2.1 Exercise 3

For the **naive Rank-Select** (RS) it stores the `bitvector` in an `int[]` array and then iterates through it to find the `rank(i)` or `select(r)`. If either `i` or `r` is < 1 or $> n - n$ being the size of the bitvector — it returns -1 in constant time, the same is true if `r` $>$ the max rank, the max `rank` being precomputed. All of the implementations are constant time for `select` in the above cases.

For **RSLookUp** it does not store the bitvector, but precomputes every `rank` at position `i` and contains that array, hence it uses the same space as the naive RS. These precomputations allow for $O(1)$ `rank` and $(log(n))$ `select` queries.

For **RSSpaceEfficient**, the bitvector is stored in 32-bit integers, each of these 32-bit blocks are split into superblocks of size

k. The superblocks holds the precomputed `rank` of all the blocks within and before it. Our implementation differs slightly in that it stores all k superblocks + 1, ensuring that in the case of k being bigger than the actual bitvector, it still holds the precomputation of the max `rank`.

The precomputation of the superblocks is done by having a table (`popc`) containing all precomputations of an 8-bit number, so each set bit in a 32-bit is counted 8-bit at the time. This allows for $O(1)$ `rank` and $O(log(n))$ `select` queries, while using much less space [González et al., 2005].

Both the RSLookUp and naive RS uses 32*n bits to store the bitvector, instead of just n bits, one way we could improve this is by changing the underlying array to `boolean[]`.

2.2 Exercise 4

We tested our implementations of the Rank - Select datastructures using both **scenario** tests and more **random** tests. Our test setup is found in the accompanying `Testing.java` class.

2.2.1 Scenario tests

For the scenario tests, four corner cases were considered, specifically the bitvectors:[1...0], [0...1], [1...1], [0...0]. All of size 64-bit. In the first and the third only one bit is a set, in the second all bits are set and in the last none are. The 5th case was a combination of all the above cases in that order. All of these cases had several `rank` and `select` queries associated with them and were stored in `.in` file which would then be held against their respective `.ans` file.

2.2.2 Random tests

The more random testing is partly inspired by property-based testing in the sense that it generates random input and tests some properties. The input is generated by providing a random value i where $1 \leq i \leq n$. n being the size of the bitvector. The properties tested are:

$$\texttt{rank(select(}i\texttt{))} = i$$
$$\texttt{rank(i)} \geq \texttt{rank(}i - 1\texttt{)}$$

This is only done for the **Naive RS**, since it will be used to test the others. These tests are a simple check of whether the two RSs get the same result on random i for both `rank` and `select`. These tests are run before every experiment and have also been run on its own. Thus we do not have an exact count for the number of random test-cases run, as we conduct $\approx$ 48.000 tests each time we run experiments or tests. This testing allows for greater coverage of our implementation and provides us with greater reliability in the results of all being correct.

2.3 Exercise 5

We used the same machine as in exercise 2.

Experimental design	
	Description
Hypothesis	We expect RankSelect-Naive to have the slowest average running time per query of the three implementations.
Performance indicators	Average running time per query
Factors	Vector size n, Seed s, Superblock size k
Levels	$n = 96,000...768,000$ by $n \cdot 2$. $s = \{3,945;34;586,478\}$. $k = \{3;10;64\}$.
Trials	200 per design point
Design points	$n \cdot s \cdot k$
Outputs	Average running time per query on a doubling vector size with three different seeds and superblock sizes.

The **algorithm parameter** k is manipulated three times to find how significant the impact of it is for the algorithm in the experiment.

The design points are set to $n \cdot s \cdot k$ to compare the average running time per query at different values of n while changing s and k. s in manipulated to limit the outliers in the generated input. Each query is repeated 200 times to provide more robust results, while a large number would have been preferred 200 * 1,500 for each `rank` and `select` queries was chosen because of a time limitation.

The input generator takes a seed, and pseudorandomly generates an array of 64 bytes longs, while also being able to make queries i where $1 \leq i \leq$ bitvector size.

2.3.1 Results

In Figure 2 only RSSpaceEfficient and RSLookUp is compared. The reason for not including naive RS is that at the smallest size of the bitvector `select` takes $\approx 93,250,000$ ns and `rank` $\approx 34,800,000$ ns, which makes naive RS so much slower than RSSpaceEfficient and RSLookUp that it is uncomparable to them, our hypotheses have been proven. However, from the experiment it shows that as k gets larger, so does the running time of RankSelectSpaceEfficient, however, even when k=3 the RankSelectLoopUp remains more efficient by a margin.

A note with regards to starting from $n = 96000$ is that we wanted to exceed the limits of the level 1 cache for at least for the naive RS and RSLookUp implementations. The doubling experiment then used to get as large n as possible. At large n the RSSpaceEfficent should out perform the others [González et al., 2005]. However, due to the naive RS being so slow the experiment was capped at bitvectors of size 768,000.

While we got the expected result, the fact that RSLookUp remained so much faster than RSSpaceEfficient is interesting, but not unexpected since it does not have to perform any loops while RSSpaceEfficient might perform some, even though they are limited.

As a last experiment we tried to break all of the different implementations, by seeing how big they could be build before getting a heap overflow error. naive RS and RSLookUp both broke at $n \approx 512,000,000$ after running for 2+ hours each. RSSpaceEfficient, however, ran for 8+ hours without causing the heap to overflow, surpassing the limits of the others.

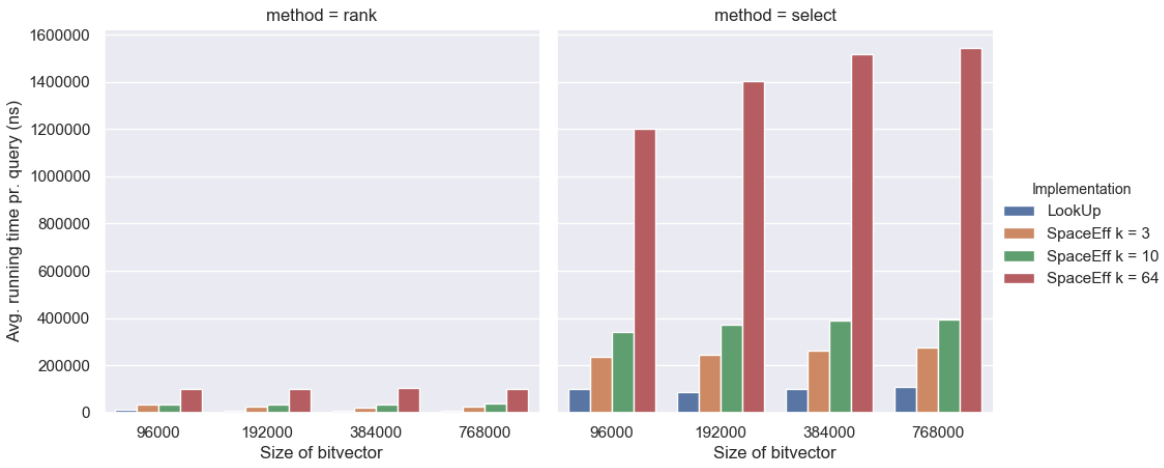

Figure 1: Distribution of all categories

References

[Brodal and Moruz, 2006] Brodal, G. S. and Moruz, G. (2006). Skewed binary search trees. In *Proceedings of the 14th Conference on Annual European Symposium - Volume 14*, ESA'06, pages 708–719, London, UK, UK. Springer-Verlag.

[González et al., 2005] González, R., Grabowski, S., Mäkinen, V., and Navarro, G. (2005). Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38.