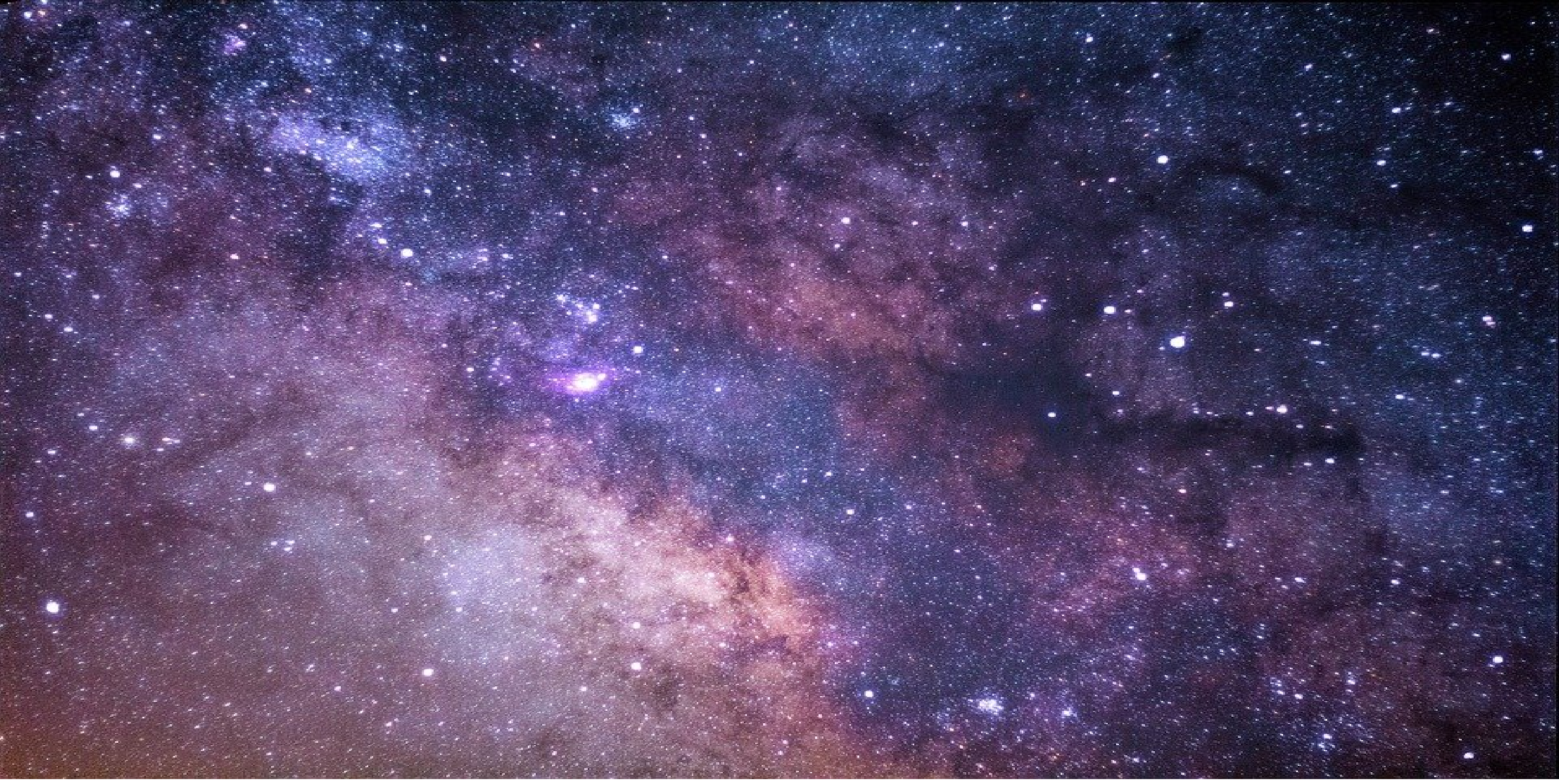


DBM-Baenk

Testing and Benchmarking environment for DBMs

Laurits Brøcker

Software, cs-22-sv-10-01, 2022-06

Master's Project

DBM-Baenk

Testing and Benchmarking environment for DBMs

Laurits Brøcker
Software, cs-22-sv-10-01, 2022-06

Master's Project

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
http://www.cs.aau.dk

Title:
DBM-Baenk

Theme:
Testing and Benchmarking of Data Structures

Project Period:
Spring Semester 2022

Project Group:
cs-22-sv-10-01

Participant(s):
Laurits Brøcker

Supervisor(s):
Ulrik Nyman

Copies: 1

Page Numbers: 32

Date of Completion:
June 8, 2022

Abstract:

This report details the process of creating a testing and benchmarking environment for several implementations of a scientific specification. Specifically, a newer generic Difference Bound Matrix (DBM) implementation is conformance-tested and benchmarked against an older, more mature DBM implementation. The usefulness of being able to benchmark multiple implementations at the same time is also shown when comparing suggested changes to the baseline implementation, in order to prevent performance regressions.

Contents

Chapter 0

Summary

This report explores the challenges and solutions to testing and benchmarking several implementations based off scientific theory. This was done by implementing a testing and benchmarking environment for two different implementations of Difference Bound Matrices (DBMs), by use of Rust metaprogramming (macros). By implementing a common interface, it is possible to generate tests in correct Rust, that is, without any compilation errors. This allows the programmer to only write a single test suite, while being able to test an arbitrary number of implementations. The discord naming schemes between theory and implementations impeded this effort, however, which further stresses the importance of documenting software well, specifically annotating a given function with its theoretical complement, so future programmers know which functions were renamed.
The importance of being able to benchmark several implementations at once shows its merit when comparing an older to a newer implementation, when merging proposed fixes, or when comparing different concrete implementations of a generic implementation, as this allows the programmer to better reason about which tool to use in a given situation.

Through the benchmarks, an interesting discovery was found about the UDBM [8] implementation. Even though the UDBM and RDBM [3] implementations share the same theoretical background [2], they sometimes differ quite wildly in runtime performance. Part of this performance difference may be explained by the use of more specialised functions in UDBM, as this implementation feature several variants of the functions specified in the theory, that are smaller in scope, but consequently also use fewer instructions, lessening the use, and thus optimising effort, for the original functions.

Chapter 1

Introduction

Testing is an important part of software development. It represents a specification for a correct program, and can be used to ascertain, that a program follows certain behaviour. This is obviously quite useful: After all, it allows programmers to verify that their code acts correctly when used in a certain manner. Not necessarily by only testing a single function call, but also testing the implementation while it is part of a larger program, using the larger program's tests to ensure correct integration in the larger codebase.
Despite the benefits, testing is sometimes left out of a software development workflow, even if only temporarily, as deadlines approach and schedules get rescheduled. Last semester, I made a reimplementation of a Differenve Bound Matrix (DBM) in Rust, namely RDBM [3], a DBM used to symbolically represent time in formal verification and model checking, and like many engineers before me, I fell victim to the classic software engineering blunder: Underestimating time use. This in turn led to omitting tests in the interest of having a more complete implementation. In this project, I seek to make amends for this decision by testing and benchmarking the RDBM library and comparing it to the UDBM library [8][4], in order to ensure compliance between the implementations and compare their resource usage.

Chapter 2

Problem Statement

Considering that theoretical computer science forms the basis upon which software engineers base their implementations, ensuring that a specification from a scientific publication has been correctly implemented is quite important. One way to do this is to thoroughly test a given implementation until the programmer is confident that most bugs have been caught and fixed. While efficient for single implementations, it can be hard to reuse a given test collection if the programmer has not exercised strict coding guidelines that help make it portable to other implementations.
A naïve solution would be to just put in the footwork and rewrite it, but considering programmers and their unnatural drive to automate manual tasks, I am confident there is a better way to do this. After all, when a data structure or an algorithm originates from a publication, there is usually a reference interface specified in the same paper.

And so, the problem statement of this project is the following:

What are the challenges involved in testing and benchmarking several implementations of a given theoretical specification, and how can these challenges be mitigated?

2.1 Testing generic implementations

Allowing benchmarks of several different implementations also has the added benefit of testing and benchmarking several different concrete implementations of a given generic implementation. The Rust language features generic structs as a zero-cost abstraction, which gives the programmer the ability to substitute the generic type for a more suitable one in any given situation, without having to change the source code of a function or struct itself. This feature isn't worth much, however, if an implementation only works with a single type. Being able to quickly generate a whole test suite for RDBM with a different generic type makes it trivial to ensure that its behaviour lives up the same specification as any other concrete RDBM type. Furthermore, if a custom datastructure for a bound is desired, it is easy to test that it performs as expected, as it merely requires an instantiation of RDBM with the type in question, to test its application in RDBM works as expected.

Chapter 3

Implementation

The testing bench is implemented using Rust macros, a form of metaprogramming, by taking a predefined set of abstract test cases, and then generating a concrete set of cases for each input type. A programmer's intuition might suggest pre- or postfixing each case with the typename or a string describing the same. In Rust macros, however, appending random strings or names to identifiers like function names is disallowed, at least out of the box.

In listing 3.1, an example of the macro used in the framework is supplied. While

Listing 3.1: The Test Macro

```rust
macro_rules! generate_tests {
    ($($name:ident: $type:ty,)*) => {
        $(
        mod $name {

            #[test]
            fn test_relation_init() {
                let x:$type = DBM::init(3);
                let y:$type = DBM::init(3);
                assert_eq!(DBM::is_included_in(&x, &y), true);
                assert_eq!(DBM::is_included_in(&y, &x), true);
            }
            // more tests go here
        }
        )*
    }
}

```

it looks intimidating at first sight, it works much like an ordinary pattern match, with the added catch that arguments can be matched upon multiple times. Going through the statement itself, macro_rules! simply tells rust we are about to declare a macro, while generate_tests is the name of the new macro. The bracket then forms the body of the macro. So far so good. The following parenthesis sets up an input to match for the macro. The implication here is, that a macro can match more than one input, although this isn't shown in the example. Then, there's a parenthesis which is prefixed with a dollarsign ($) and postfixed with a star (*). This signifies that the contents of the parenthesis can be matched zero or more times. Inside the parenthesis are two parameters that are matched on: `$name`, an identifier (denoted by `:ident`), and `$type`, a type (denoted by `:ty`). The ⇒ leads to the output code, where we once again see the `$()*` syntax, though this time, denoting the resulting code from each match in the macro call. In practical terms, this code snippet would then generate a new module for each (name, type)-pair, with the tests specified inside the module.

While packages like Paste [10] do exist and can provide functionality for generating function names from a string or identifier, Eli Bendersky[1] formulated this simpler way, taking advantage of the Rust scoping rules. Rather than using any sort of pre/postfixing, the identifier instead declares a module in which all the tests will be generated. This way, there will also not be any duplicate test names, as the functions are declared in different modules, and as such, different scopes.

In order to provide a common interface for DBMs to implement, the DBM trait is used to ensure compliance across implementations. In listing 3.1, this can be seen in the function calls `DBM::init` and `DBM::is_included_in()`.
Granted, it isn't *strictly* necessary to use a trait for this, as implementations merely need a function with the same signature as the one being called, in order to be used in the macro.
Regardless, I chose to use a trait anyway, as it makes it easier to get an overview of what needs to be implemented for the test suite to work. This does, however, also mean that all calls to the trait functions must be quantified as such, rather than just calling them on the value, as some implementations may have the same function names as those present in the trait. For example, it might seem natural to call `$type::init(10))` to instantiate a type, but if the underlying type already has an associated function called `init`, that implementation will be called rather than the `DBM::init` implementation, or if the parameters or return values are different, result in a type error upon compilation.

Implementing the DBM trait for a DBM-implementation is fairly straightforward. First, take a look at the trait definition in listing 3.2.

These functions should be implemented over an existing struct, as can be seen in listing 3.3, which shows an implementation for the RDBM struct. Here it can

Listing 3.2: Definition of DBM trait

```
pub trait DBM<T> {
    fn init(dim: usize) -> Self;
    fn zero(dim: usize) -> Self;
    fn is_included_in(rhs_dbm: &Self, lhs_dbm: &Self) -> bool;
    fn is_satisfied(dbm: &Self, i: usize, j: usize,
        bound_is_strict: bool, constant: T) -> bool;

    fn close(dbm: &mut Self);

    fn future(dbm: &mut Self);
    fn past(dbm: &mut Self);
    fn restrict(dbm: &mut Self, i: usize, j: usize,
        bound_is_strict: bool, constant: T);
    fn free(dbm: &mut Self, clock: usize);
    fn assign(dbm: &mut Self, clock: usize, constant: T);
    fn copy(dbm: &mut Self, clock_to: usize, clock_from: usize);
    fn shift(dbm: &mut Self, clock: usize, shift_constant: T);
}

```

Listing 3.3: Partial implementation of DBM trait for RDBM

```
impl<T: std::ops::Neg<Output = T> + Zero + Bounded + Clone
+ Ord + num::Saturating> DBM<T>
for RDBM<T>
{
    fn init(dim: usize) -> RDBM<T> {
        return rdbm::DBM::new(dim);
    }

    fn zero(dim: usize) -> RDBM<T> {
        return rdbm::DBM::zero(dim);
    }

    fn is_included_in(lhs: &RDBM<T>, rhs: &RDBM<T>) -> bool {
        return rdbm::DBM::is_included_in(lhs, rhs);
    }
    fn is_satisfied(dbm: &Self, i: usize, j: usize,
    is_bound_strict: bool, constant: T) -> bool {
        let constraint_op = match is_bound_strict {
            true => rdbm::ConstraintOp::LessThan,
            false => rdbm::ConstraintOp::LessThanEqual,
        };
        return rdbm::DBM::satisfied(dbm, i, j,
        constraint_op, constant).unwrap();
    }
    // Functions below omitted in listing for brevity's sake
}

```

also be seen how the trait serves as an abstraction layer over the more specific details like `ConstraintOp` in RDBM [3], which is then interpreted as a bool in the trait function.

Considering both trait implementations and macros, it is now possible to make a generalized macro that generates tests reliably for structs that implement the DBM trait.

3.1 Conformance testing

A huge benefit that can be derived from the macro-based approach to test generation is the ability to assert the degree to which two or more implementations conform to the same specification, while still being (relatively) painless to implement for an arbitrary DBM. The most obvious way to perform this conformance testing is by comparison to another implementation. In this case, comparing RDBM to UDBM. The other, more subtle approach is to conformance test a custom type to denote a bound by comparing RDBM to itself, but with two different generic types. The following section will touch on both.

3.1.1 Conformance testing with UDBM

Since there already exists a DBM-implementation in UDBM, we can leverage this to our advantage. Rather than just generating tests for RDBM, the same tests are then generated for UDBM, in order to ensure that our tests are correctly written. That is, if a test doesn't pass when using UDBM, it is most likely incorrectly written, as UDBM is quite old and mature, and therefore assumed to be more thoroughly tested. Naturally, it could also be the UDBM wrapper which contains an error, though these are relatively uncommon, as the rust functions map almost directly to their C++ counterparts.
Speaking of the UDBM wrapper, it is made using Bindgen [7], a Rust tool for generating Rust source files from C/C++ headers, so that the Rust code calls a foreign function through the C FFI (Foreign Function Interface). One issue that appeared while making this wrapper was the liberal use of inline functions and preprocessor directives, that didn't play nicely with foreign calls. Granted, this *does* make sense, as both inline functions and pp-directives specifically *don't* generate a function signature to be called. The workaround I decided on was to make my own C++ wrapper with complete function signatures, that then have their rust bindings generated by Bindgen.
Another challenge that appeared when writing the wrapper was the disconnect in the naming scheme between the functions originally specified in *Timed Automata: Semantics, Algorithms and Tools* [2], and the actual functions implemented in Uppaal

TA:SAT [2]	UDBM	RDBM	DBM-trait
unspecified	dbm_zero	zero	zero
unspecified	dbm_init	new	init
relation	dbm_relation	is_included_in	is_included_in
satisfied	dbm_satisfies	satisfied	is_satisfied
close	dbm_close	close	close
up	dbm_up	up	future
down	dbm_down	down	past
and	dbm_constrain1	and	restrict
free	dbm_freeClock	free	free
reset	dbm_updateValue	reset	assign
copy	dbm_updateClock	copy	copy
shift	dbm_updateIncrement	shift	shift

Figure 3.1: Table showing the different names in their respective namespaces

itself. This made it somewhat harder to know which functions should be bound, as they aren't always straightforwardly named, though with enough code digging, it certainly is possible.

Two notes should be made about the table in figure 3.1. First, the original paper doesn't specify any initialising function, whereas both UDBM and RDBM have one. For this reason, it also made sense to add to the trait: Otherwise, how would one even instantiate a DBM? The other note is on the relation/included split. In the original paper and in the DBM library, the relation function performs an inclusion check both ways between the DBMs through a single pass of the DBM, returning a number that symbolises either $A \subset B$, $B \subset A$, $A = B$, or $A \not\subseteq B \land B \not\subseteq A$, where A and B are the two DBMs used as input. RDBM, however, just checks if the first DBM is included in the other, that is $A \subseteq B$. This is due to the simpler parsing of the result, which in turn, makes it easier to write code for, especially though the FFI. For linking with UDBM, this would require parsing the status codes to Rust too, which is doable, but takes time. In contrast, the inclusion check merely returns a bool, something well supported by the FFI and several languages alike, so this makes the implementation of this function for any future DBMs relatively simple.

3.2 Benchmarking

Part of the original [3] design goals of RDBM was to make a DBM-implementation that was comparable in performance and memory use to UDBM. Luckily, we can reuse the trait implementation from the conformance tests to compare the runtime of the different implementations. In order to perform this benchmark, I've decided

Listing 3.4: A snippet of the benchmarking macro

```rust
macro_rules! generate_benchmarks {
    ($($name:expr, $type:ty,)*) => {
        pub fn zero_benchmark(c: &mut Criterion) {
            let mut group = c.benchmark_group("Zero");
            for i in [20u64, 100u64].iter() {
                $(
                group.bench_with_input(BenchmarkId::new($name, i),
                i, |b, i| b.iter(||
                {let _x:$type = DBM::zero(*i as usize);}));
                )*
            }
        }

        pub fn init_benchmark(c: &mut Criterion) {
            let mut group = c.benchmark_group("Init");
            for i in [20u64, 100u64].iter() {
                $(
                group.bench_with_input(BenchmarkId::new($name, i),
                i, |b, i| b.iter(||
                {let _x:$type = DBM::init(*i as usize);}));
                )*
            }
        }
    }


```

to use Criterion [9], a Rust library for performing benchmarks. The benchmarks are also made using a macro, as this, once again, allows us to compare the performance of different implementations in a common benchmark. An example of this can be seen in listing 3.4.

Much like the test macro, pattern expansion is used to substitute the concrete types, though instead of generating one big module for each case, Criterion knows how to group the benchmarks, and so the expansion is just registering a function from each implementation in its corresponding benchmark.

The scrutinous reader will notice that the benchmarks are grouped by an input too. This input represents the dimension of the DBM: That is, the number of clocks a DBM represents at once. I won't provide an in-depth explanation in this paper,

but I will instead refer to [2], in which an excellent explanation is given.
For the scope of this paper, the only relevant information is, that a DBM is a square matrix, where the side length is equal to the number of clocks the DBM is supposed to represent. A three-dimensional DBM refers to a DBM in which the side length, i.e, number of clocks is three, and thus is composed of nine bounds.
This means that as the dimensions grow, so do the elements the algorithms have to check, which makes it a good candidate for measuring runtime against.

3.2.1 Performance improvements

Now that the benchmarks have been set up, they can be used to compare DBM implementations. This is not limited to only UDBM, but also different versions of RDBM, either with different concrete types, or an older RDBM version compared to a newer one. This is quite useful for comparing the runtime difference between using 32-bit integers and 8-bit integers, tracking performance improvement over time, or ensuring that a proposed code merge does not cause a performance regression.

Specifically, I had two performance improvements in mind for RDBM back when I finished that project [3]. The first one is keeping track of clocks with a single integer value, rather than using a list of clock names. Back when I made that choice, it seemed sensible, but obviously, hindsight is 20-20.
The other performance improvement is using another bitvector implementation. I made my own as an onboarding project to Rust, and thus, I am not particularly confident in the performance of that code.
Luckily, both of these can be tested relatively easily, as I simply import them as a git-dependency in my Cargo.toml file, and set their respective branches. Then I implement the DBM trait for the structs, and add them to the benchmark macro to compare them.

3.3 Benchmark results

The results of the benchmark generally shows that RDBM features comparable performance to UDBM. Though I ran the benchmarks for all the functions specified in the DBM trait, I only picked out a couple of the most interesting. Do note that the x-axis in the following graph represents dimensions, even though the label says "input". The y-axis is the average runtime. As such, this means the graphs show how the aveerage runtime grows as dimensions grow. Furthermore, I've added several other versions of my implementation as comparison. The rdbm_v1 version represent the library without any optimisations, and only bugfixes that I found while developing the test bench. It represents the RDBM library as I left it from my previous report [3]. This is used as a baseline for comparison with

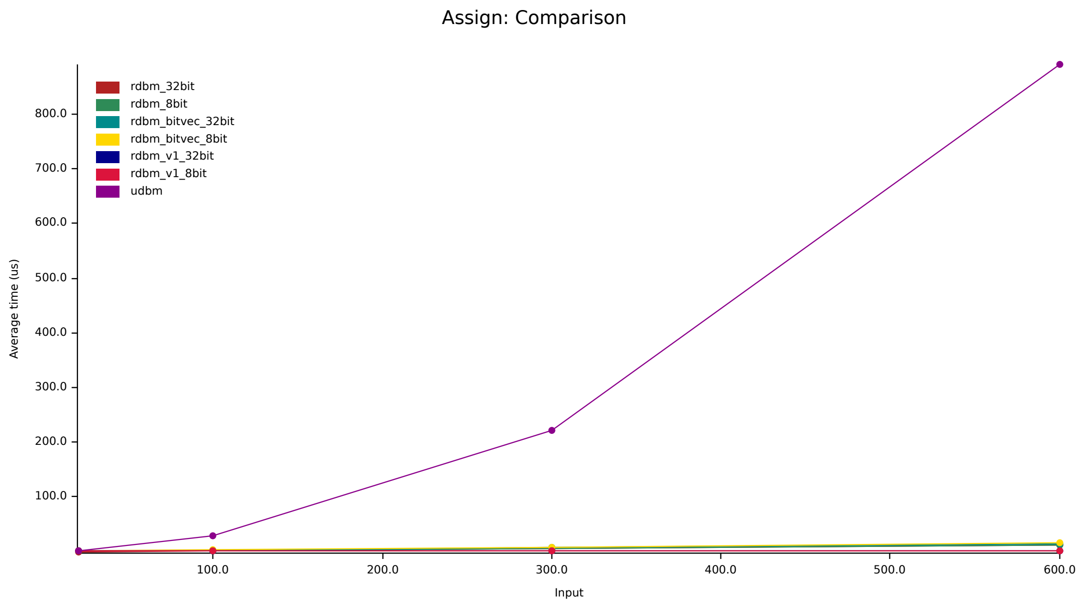


Figure 3.2: A graph showing the average runtime per dimensions for the Assign function

the plain rdbm version, in which I changed the clock name behaviour mentioned in section 3.2.1. Lastly, the rdbm_bitvec version represent a proposed change to how bitvectors are implemented (also mentioned in section 3.2.1), to see how this change would impact the runtime. For these three RDBMs, I've also elected to benchmark their performance using two different generic types, one with 32-bit integers, the other with 8-bit integers. They are denoted with _32bit and _8bit in the graphs respectively.
Lastly, any graph shown in this section can be found in appendix A.1

Sometimes, like in the Assign-benchmark (fig. 3.2, app. A.1.1), RDBM handily outcompetes UDBM. It appears as if the runtime in UDBM grows exponentially, whereas the RDBM runtime can barely be seen along the x-axis. This is quite surprising, as both RDBM and UDBM are based on the same theory [2] and implemented in languages that are roughly as fast as each other.

The same pattern can be seen in figure 3.3, where RDBM also outcompetes UDBM.

That being said, in the Restrict-benchmark (fig. 3.4, app. A.1.3), UDBM comes out on top, even if it isn't much in absolute terms, so it isn't always the case that RDBM is faster than UDBM.

One extraordinary benchmark is the Close benchmark (fig. 3.5, app. A.1.4), as UDBM completes the benchmark in a very short amount of time. This isn't consistent with its description in [2], where it is described as an expensive operation.

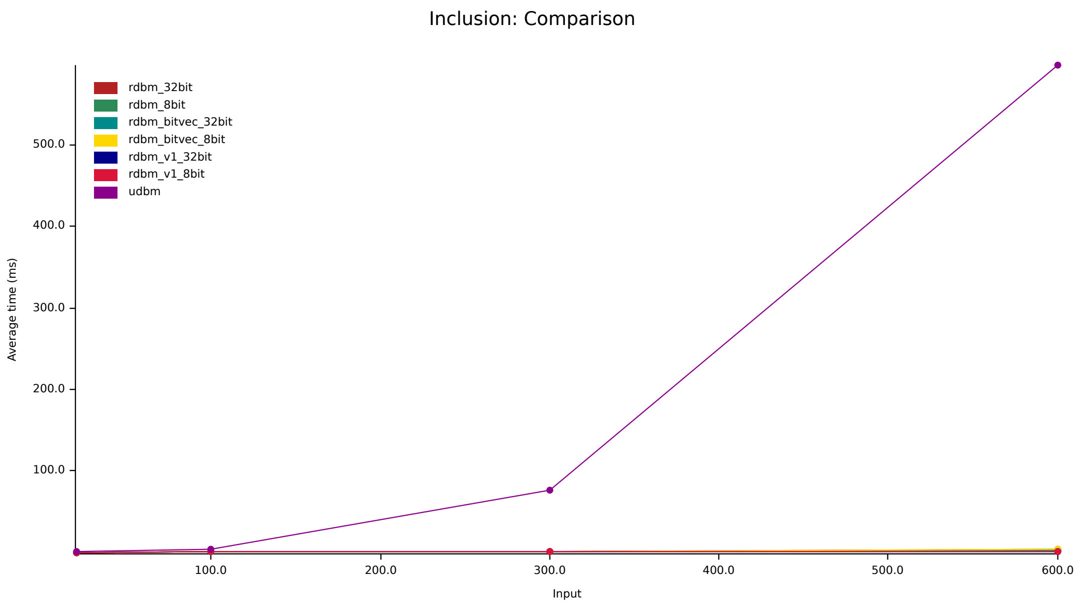


Figure 3.3: A graph showing the average runtime per dimensions for the Inclusion function

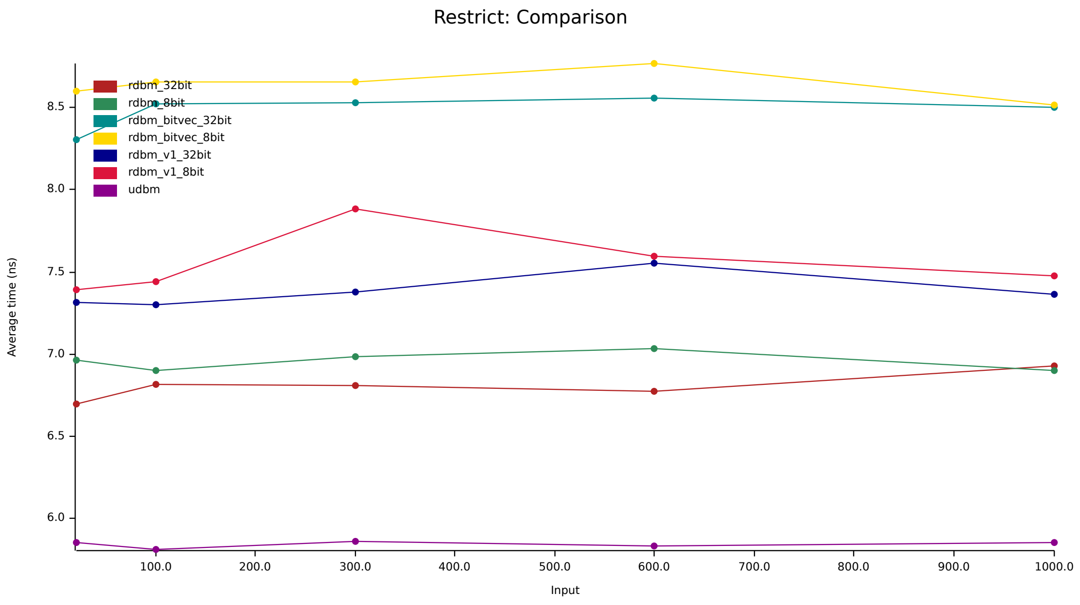


Figure 3.4: A graph showing the average runtime per dimensions for the Restrict function

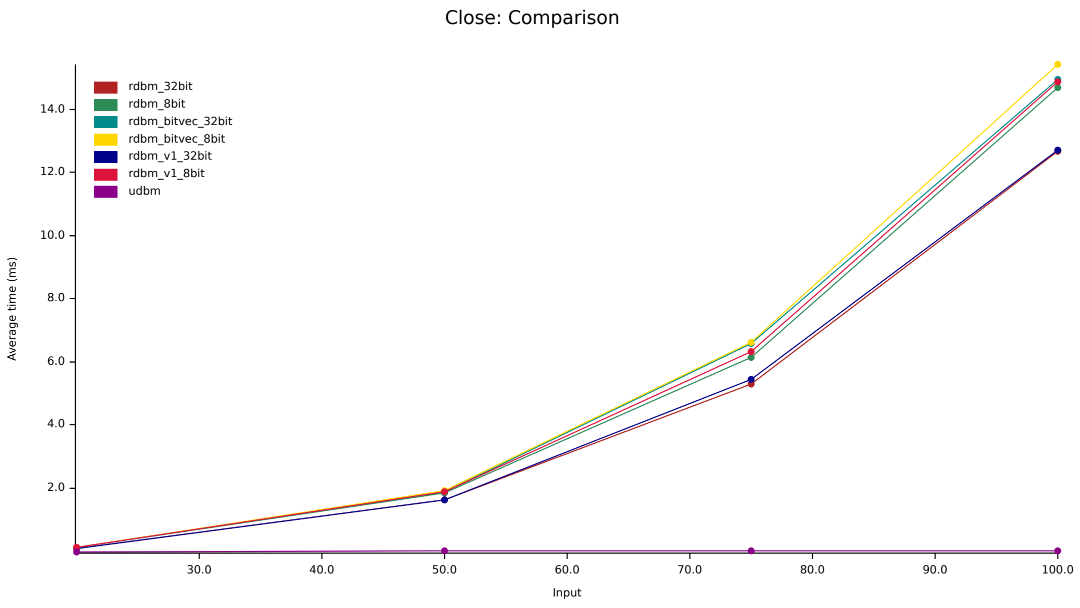


Figure 3.5: A graph showing the average runtime per dimensions for the Close function

This leads me to believe that the UDBM implementation of Close has a necessity check, which only runs the function if the DBM isn't already closed. Therefore, I would not consider the UDBM observations in this benchmark indicative of its actual performance.

In the following graphs, UDBM has been omitted, as it makes it hard to discern between the different DBM implementations. Furthermore, due to time constraints, I also didn't add the 8-bit versions.

The general tendency in these benchmarks is, that the plain rdbm sometimes beat the bitvec version (fig. 3.7, app. A.2.2), though it isn't a guarantee, as the bitvec version sometimes outperform the plain version (3.6, A.2.1). The v1 version behaves in an unexpected manner at the 300-dimension mark, and as well or worse than the plain version until that point. Presumably, this is due to the clock indicies being tracked with an unsigned 8-bit integer in the v1 version, which results in faulty behaviour once dimensions grow beyond 255, though interestingly enough, it didn't seem to affect the runtime of the Restrict benchmark (fig. 3.8, app. A.2.3), unlike the other benchmarks.

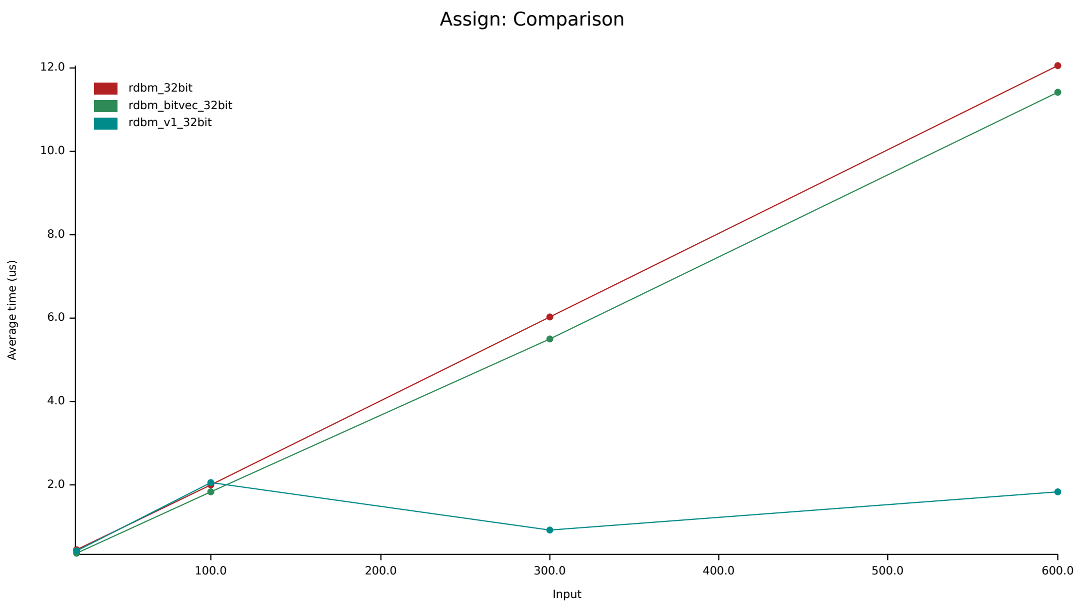


Figure 3.6: A graph showing the average runtime per dimensions for the Assign function

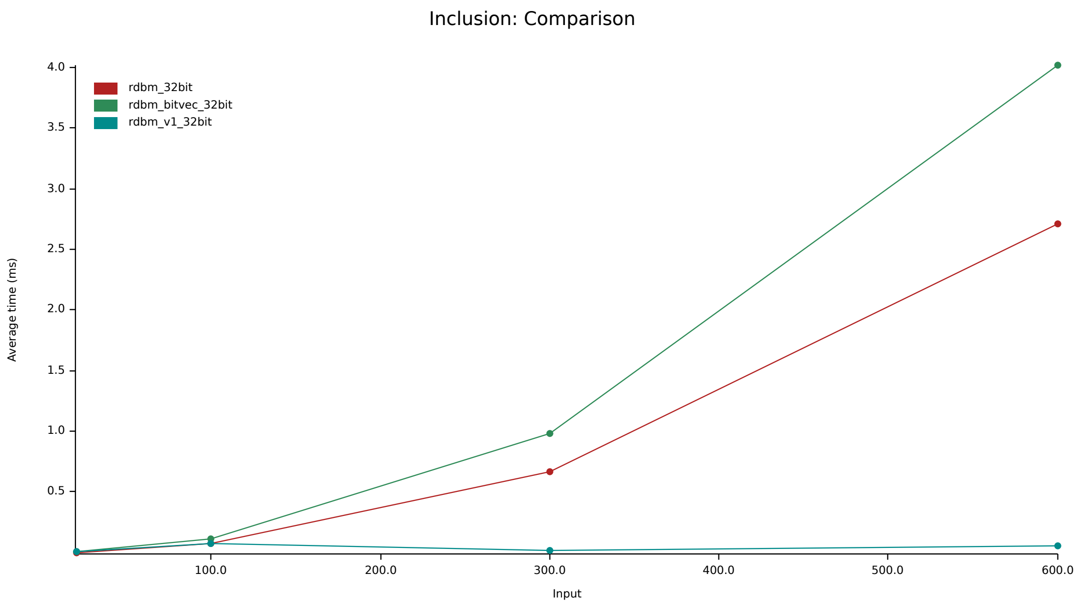


Figure 3.7: A graph showing the average runtime per dimensions for the Inclusion function

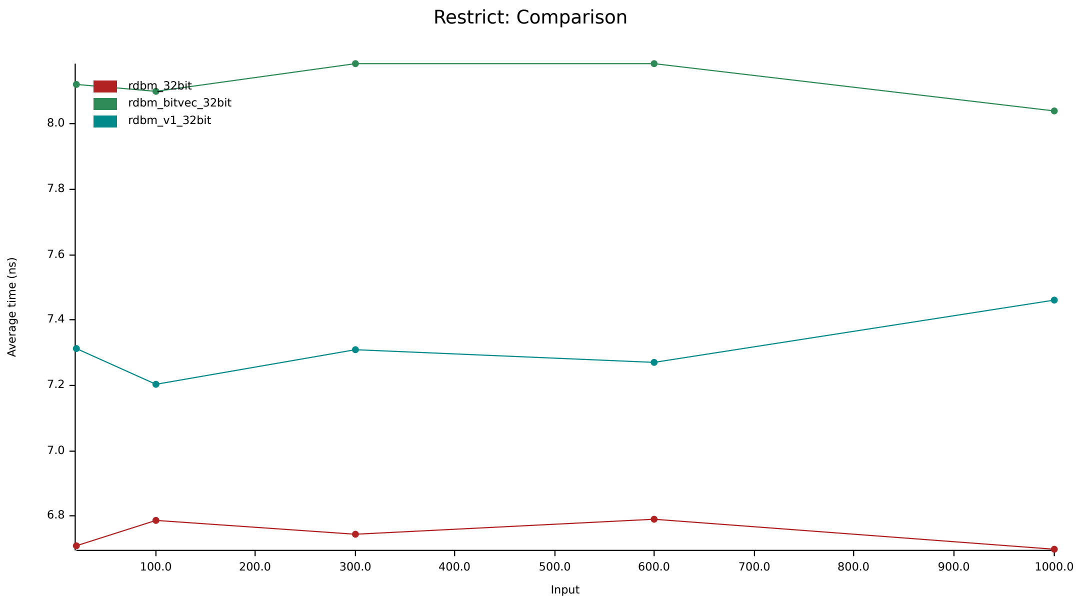


Figure 3.8: A graph showing the average runtime per dimensions for the Restrict function

Chapter 4

Discussion

This chapter contains some of the criticisms or shortcomings of this project that I wanted to address before concluding on what has been learned throughout the entire process.

4.1 Use of non-specialised functions from UDBM

Currently, I benchmark the functions specified in [2], and then consqeuently implemented in UDBM[8] and RDBM[3]. While this is arguably the only way I can properly compare the two implementations, as I didn't have the time to make a feature-complete implementation of UDBM, it also means that some functions in UDBM may be more suited in practice for a given usecase. Considering a function like Close, UDBM has a generalised dbm_close function, just like RDBM, however a quick look in the UDBM Programmer's Manual [4] will reveal several other close-implementations like close1, closex and close_ij. This benchmark does not take alternate versions into consideration, meaning it isn't always indicative of actual performance where an alternate version has been used.

Likewise, if the generalised versions aren't actually used that much in practice, the UDBM programmers likely haven't invested a lot of time in optimising these functions, instead focusing on the ones that are called more, and thus matter more for runtime.

4.2 Lack of memory benchmarks

The astute reader might have noticed the lack of memory benchmarks in this project. Especially considering that memory benchmarks were a part of the "Future Works"-section in my RDBM-report [3].

I wanted to include some, just to get an idea of the memory footprint between the two implementations, and Rust does feature a *size_of* function, but the issue I ran into is, that *size_of* doesn't measure size deeply, i.e. given a pointer, only the pointer is measured, not the memory behind it. This does make sense, as Rust doesn't have a way of knowing exactly how much memory is allocated behind the given pointer, so each implementation would need to implement their own *bytes_allocated* function if this approach is to work. While it is doable, I don't consider it a clean implementation. What if the programmer makes a mistake? Especially if the DBM is composed of more than just an array with a known element size, this becomes a tedious task for the programmer.
The other approach I thougth about was replacing the default allocator in Rust with Jemalloc [5] through the Jemallocator [6] crate, and then measuring the heap size before and after allocating a DBM, meaning the difference is the actual size of a DBM. Works for any DBM-implementation, doesn't require any extra functions, so it should be fine, right? My issue with this approach was purely technical, as I found out that Jemalloc wouldn't build if the path to the project directory contained a space. I value the ease of use of the bench, meaning that build failures due to arbitrary circumstances like a directory path are a deal breaker for me. Testing and benchmarking should be easy to perform, otherwise they will easily be forgotten. Some developers will likely consider this stance petty; That is fine, they are more than welcome to implement this approach themselves, but I will not.
The final approach is to use a 3rd-party program like Valgrind to perform the benchmarking. This would require Valgrind be installed on the host machine, complicating the installation procedure. As mentioned previously, I consider ease-of-use a very important metric for testing benches, and as such, this is not acceptable to me.

I did, however, make a couple manual memory measurements using *size_of* on both RDBM and UDBM, just to measure the current memory overhead. As it turns out, Rust vecs have a relatively large overhead for this usecase, as each vec features a capacity[1], a size[2] and a pointer to the data itself. Each of these are the size of a pointer, meaning 64 bits on a 64-bit machine, or 8 bytes. My UDBM wrapper uses a vec to allocate behind the scenes, leading to a little more memory overhead than the C++ version, where the programmer has to allocate the memory themselves, as well as keep track of dimensions in a seperate variable. By contrast, the vec I used is 24 bytes, with an added 8 on top, as I have a seperate dim variable to keep track of dimensions, though it could be boiled down to the 16 bytes you would usually see in C++ (pointer and dimension combined). RDBM by contrast uses two vecs in addition to a seperate dim value. This is quite a bit more than UDBM,

[1]number of elements the vec can hold without reallocating
[2]the actual number of elements in the vec

as the two vecs by themselves represent 48 bytes, and then an added 8 for the dim on top for a combined 56 bytes overhead. The good news are that this value can be reduced drastically by allocating the memory manually in Rust, as the capacity and size values are completely redundant as dim covers both. Furthermore, it can be shared between the two vecs, bringing the potential memory overhead down to 24 bytes.
The downside would be losing all the list functions that come with Rust vecs, meaning they would need to be reimplemented on a per-need basis. Once again, very doable, but takes it time, and as the project progressed, I found myself short on it.

Chapter 5

Conclusion

In chapter 2 I posed the following question:

What are the challenges involved in testing and benchmarking several implementations of a given theoretical specification, and how can these challenges be mitigated?

My answer is the following:

The two big challenges are increased programmer workload and inconsistent naming schemes.
The way to mitigate these problems are to use metaprogramming to generate the test cases for each implementation, and specifically for Rust, leveraging the module scoping system to keep the tests for each implementation in seperate namespaces. Regardless of implementation language, care must be given to ease-of-use, as this is a primary concern for any testing/benchmarking bench, and any regression in usability must be carefully considered before committing to it.
As for the naming schemes, if a name is ambiguous, even in context, renaming isn't *that* bad, though for future programmers' sake, make a note in the source code or documentation specifying that function X is an implementation of function Y in a given paper/implementation.

5.1 Future work

This section contains suggestions for future work on this project.

5.1.1 More tests

It is common knowledge that a test suite is never really done, it is just that writing tests offers an increasingly diminishing return compared to the effort put in. For this project specifically, I had to strike a balance between not only tests, but also benchmarks and report. While I wouldn't say that the current test suite is inadequate, it could definitely still be more comprehensive.

5.1.2 More benchmarks

Like with testing, more benchmarks are desirable. This could either be as an entirely new benchmark, or just adding more benchmarks for a specific number of dimensions. They were omitted in the project's current state, as they already take quite a while to perform, and as I don't have dedicated hardware for benchmarking, my computer is practically unusable while the benchmarks run, due to the noise it would induce in the tests.

5.1.3 Continuous integration

CI is a relatively new practice where a series of code checks are performed upon committing to a git repository. Putting CI in perspective to this project, it would make sense to have a some sort of integration with a dedicated benchmarking server that simply runs the benchmarks on any proposed changes to the code, and reports back what the performance characteristics are, as a way to provide developers with feedback on their code performance without them having to perform the benchmarks on their own machines.
This won't work with an ordinary CI-pipeline, however, as several CI-pipelines tend to be handled by the same scheduler, which would lead to noisy benchmarks. Simply put, it would need to be scheduled on dedicated hardware.

5.1.4 More DBM-implementations

Comparing RDBM and UDBM is a fine start, but they only represent two different implementations. Ideally, more should be added to the tests and benchmarks in order to create a more comprehensive perspective on the state of DBM compliance/performance.

5.1.5 Better documentation

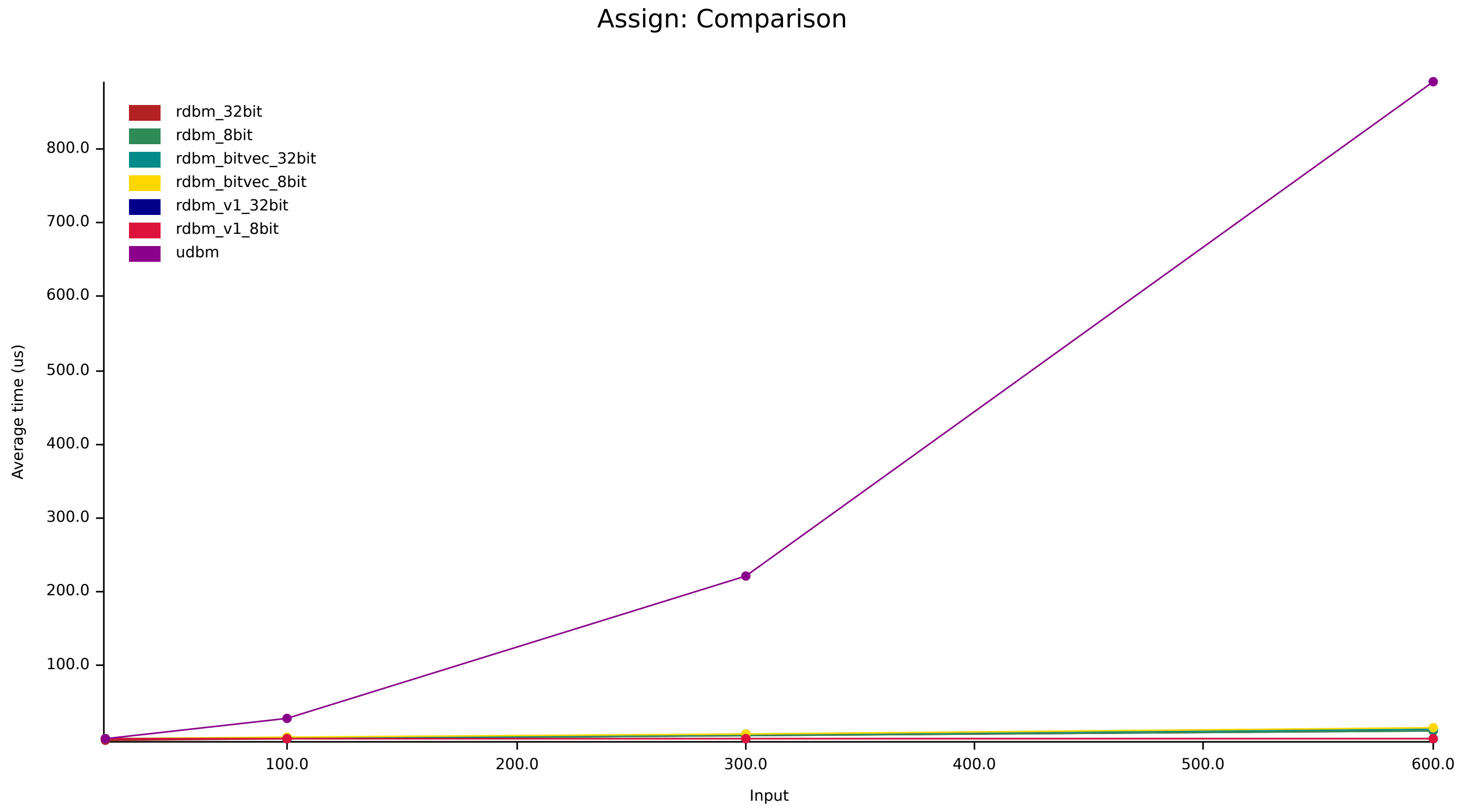
In a sense, this report is documentation for DBM-Baenk, but otherwise, not much care has been taken to ensure good documentation, except for some comments in the most arcane parts of the code. This should probably be fixed.

Appendix A

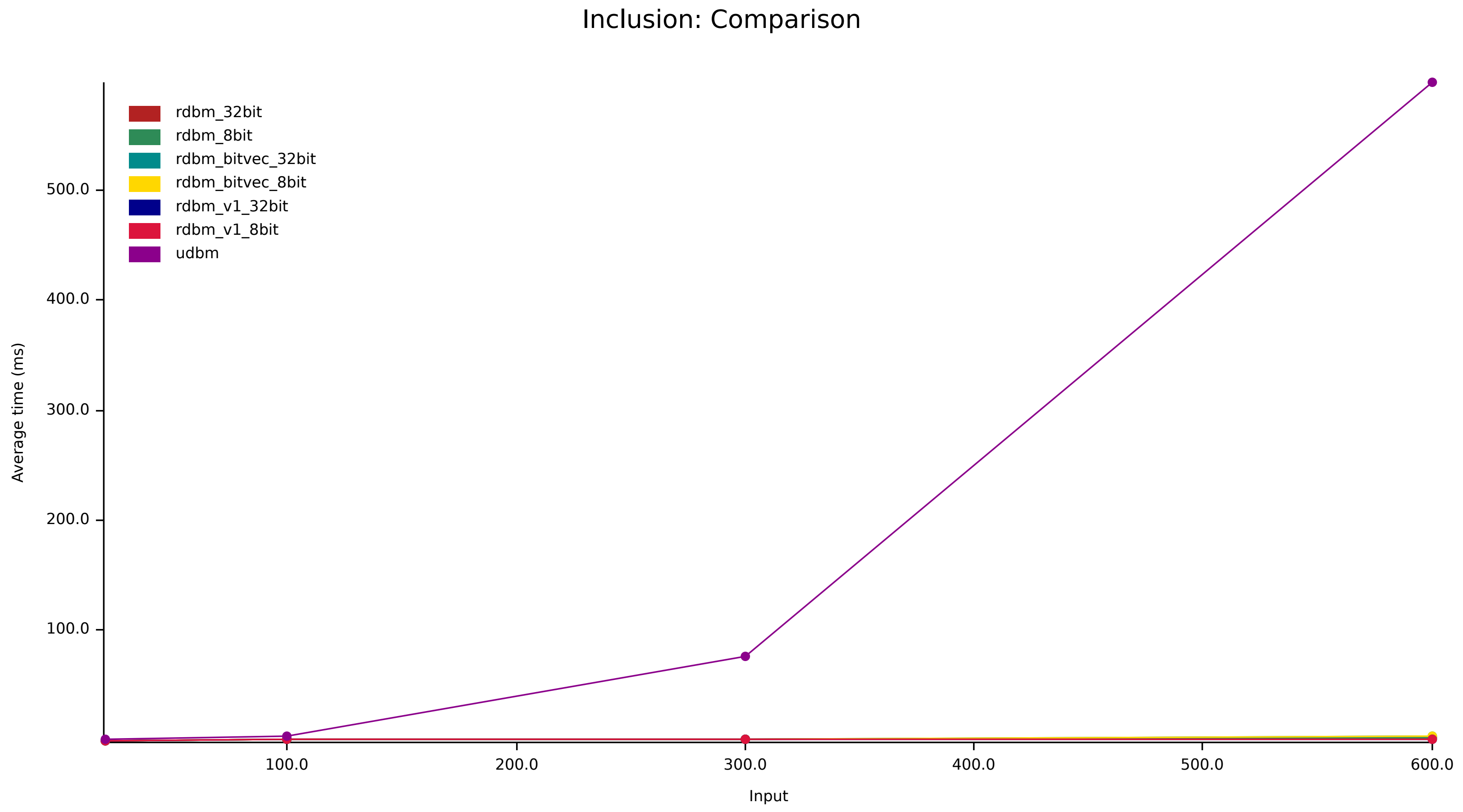
Performance graphs

A.1 Comparisons with UDBM

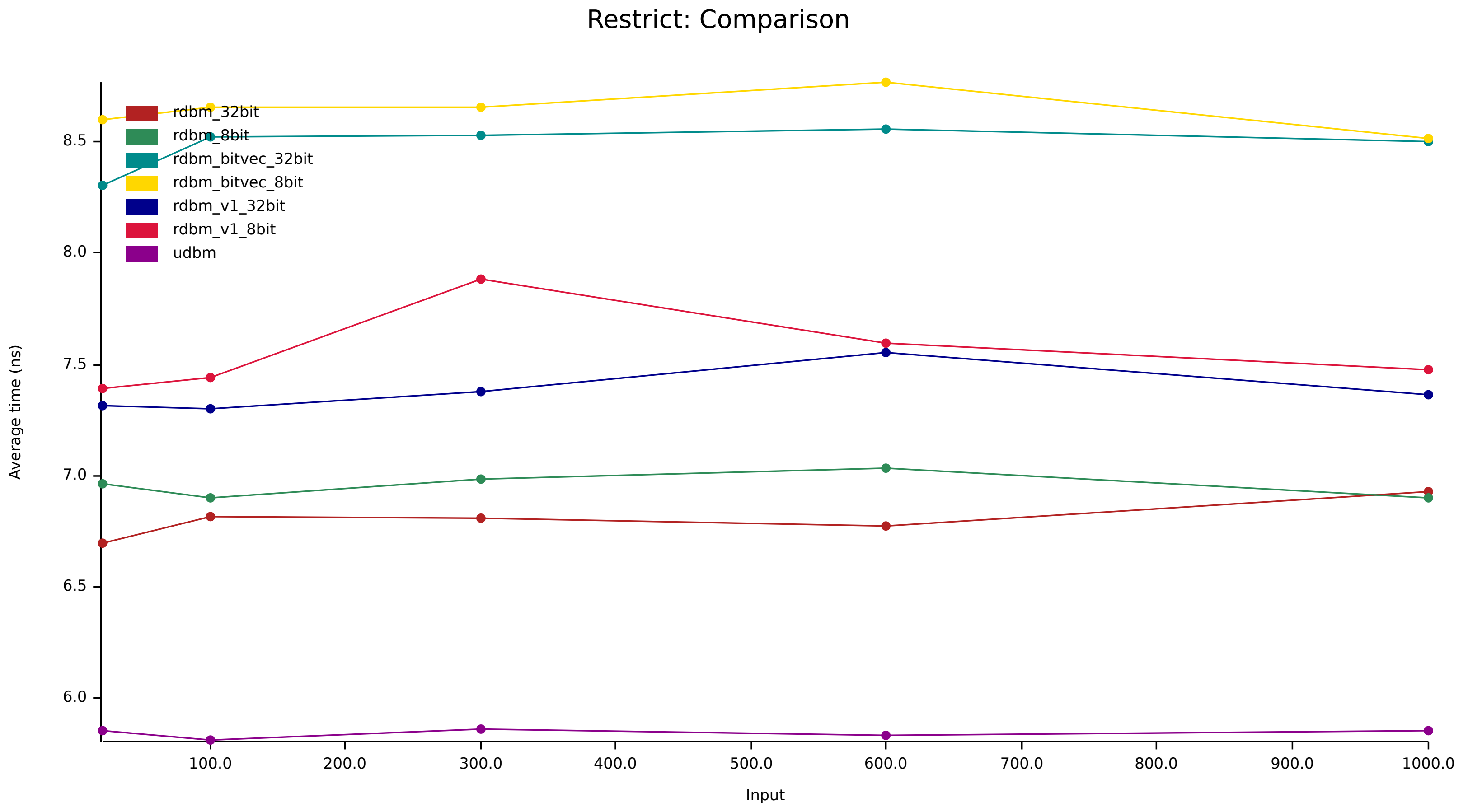
A.1.1 Comparison of Assign-function



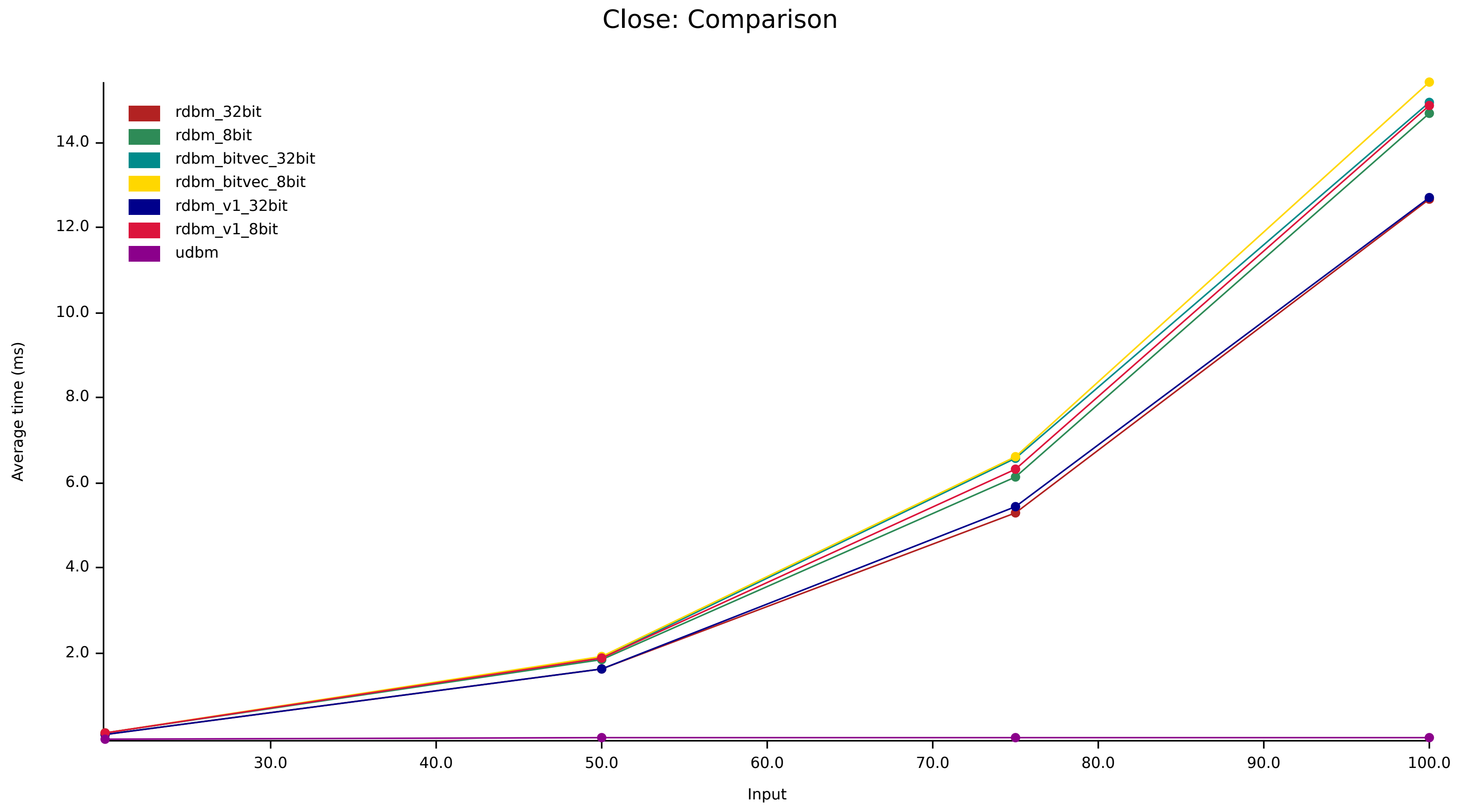
A.1.2 Comparison of Inclusion-function



A.1.3 Comparison of Restrict-function

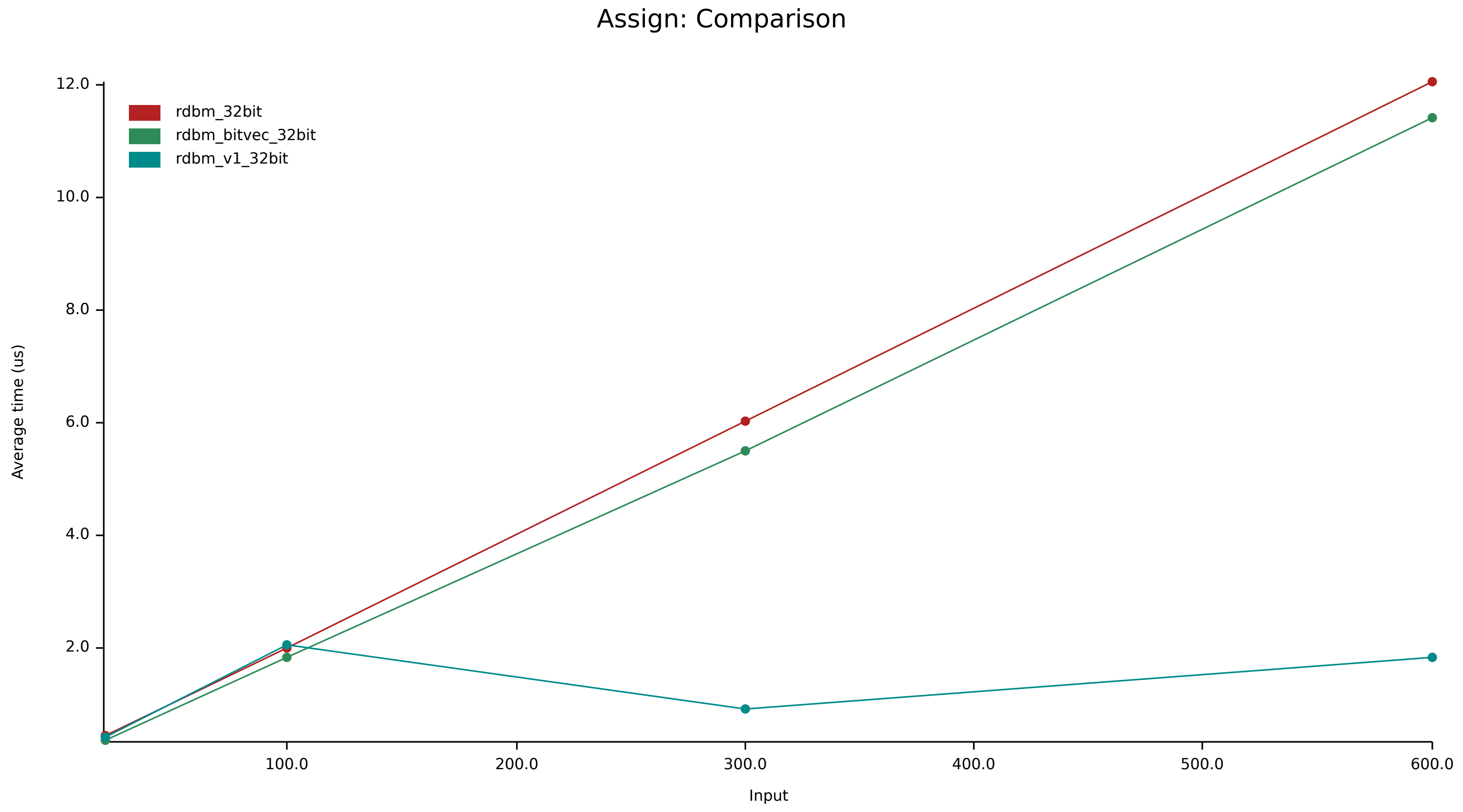


A.1.4 Comparison of Close-function

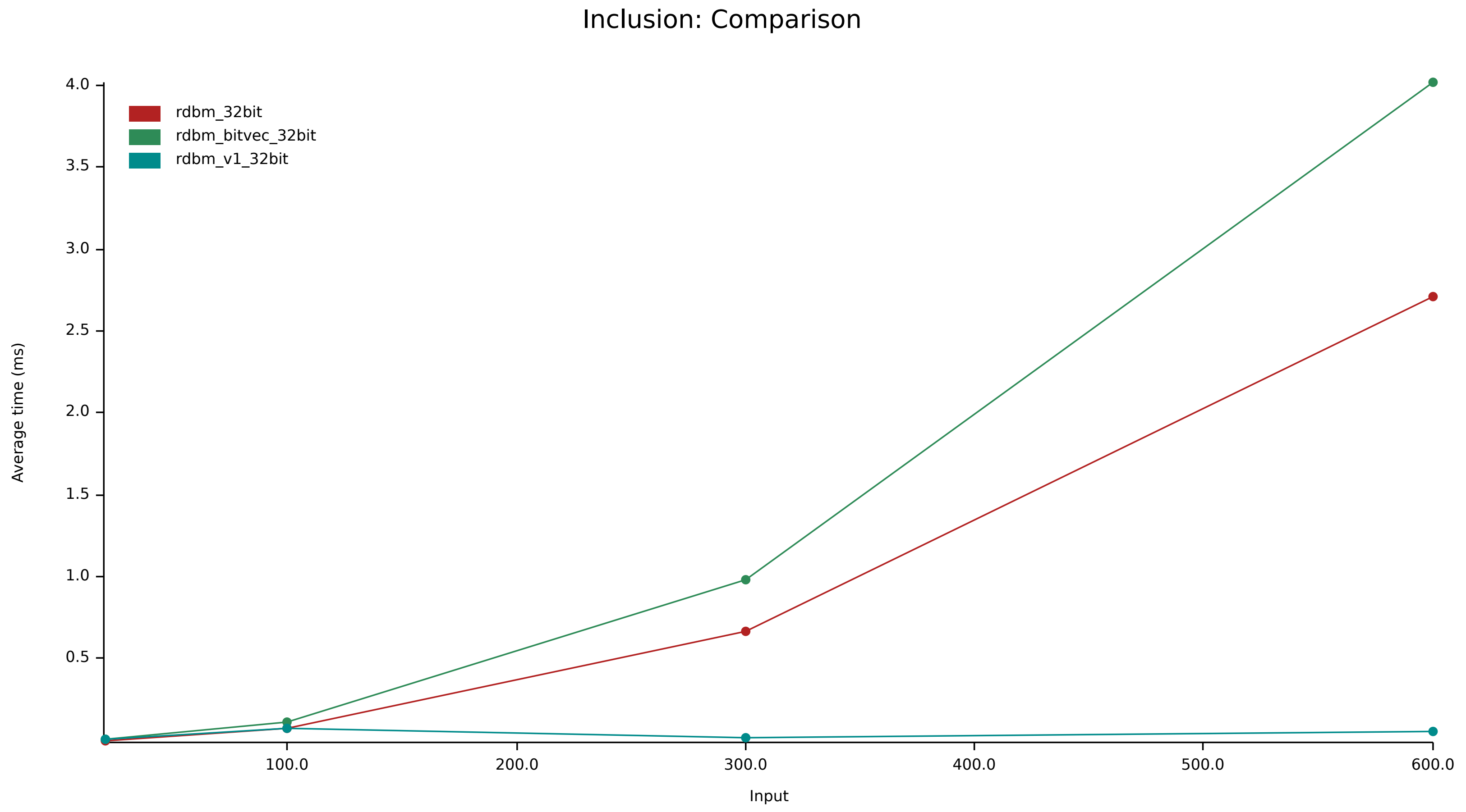


A.2 Comparisons between RDBM versions

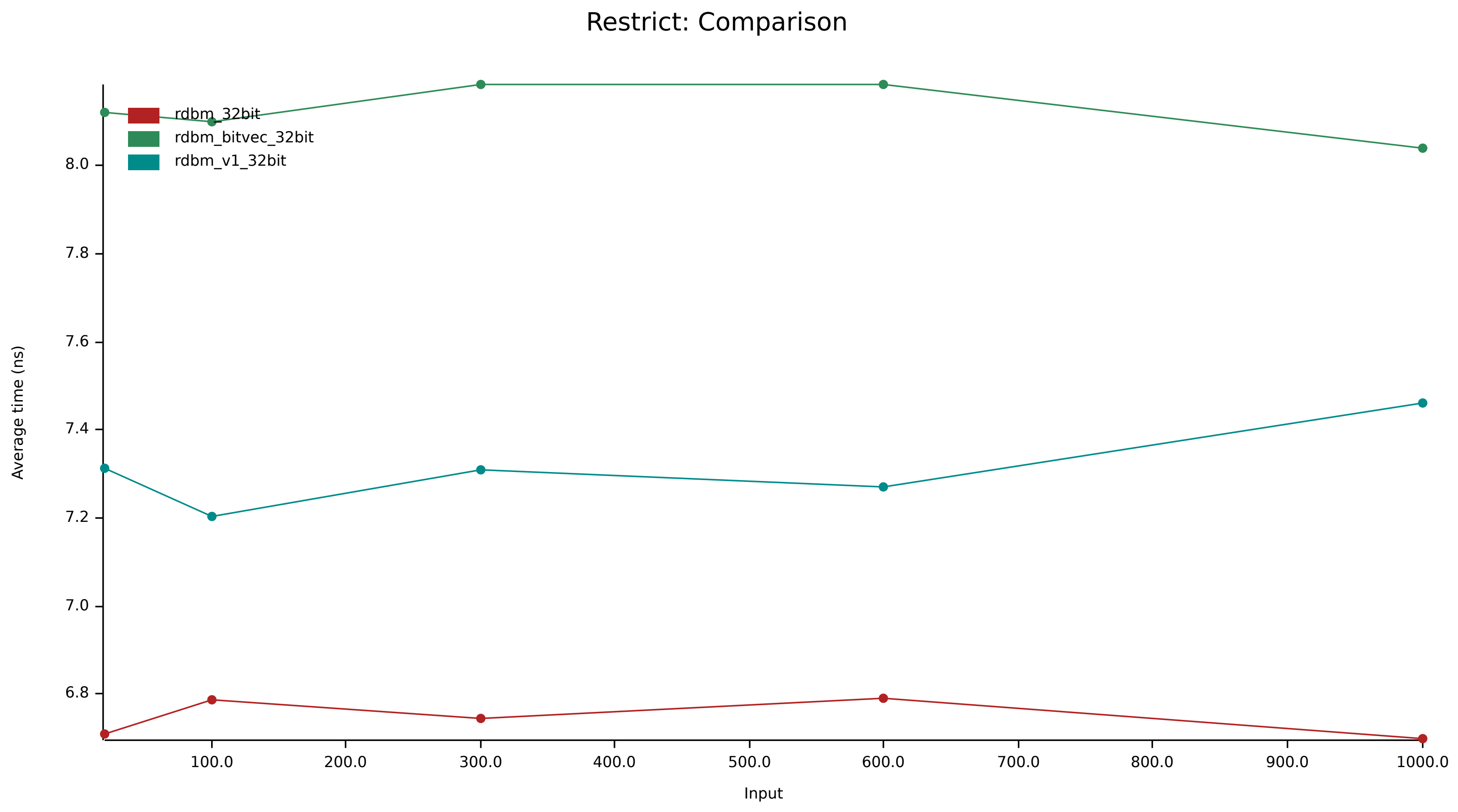
A.2.1 Comparison of Assign function



A.2.2 Comparison of Inclusion-function



A.2.3 Comparison of Restrict-function



Bibliography

[1] Eli Bendersky. *Testing multiple implementations of a trait in Rust*. URL: `https://eli.thegreenplace.net/2021/testing-multiple-implementations-of-a-trait-in-rust/`.

[2] Johan Bengtsson and Wang Yi. "Timed Automata: Semantics, Algorithms and Tools". In: Springer-Verlag, 2004, pp. 87–124.

[3] Laurits Brøcker. *RDBM. Implementation of a memory safe DBM library*. Pre-master's project. Aalborg University.

[4] Alexandre David. *Uppaal DBM Library Programmer's Reference*.

[5] Jemalloc Developers. *Jemalloc*. URL: `https://github.com/jemalloc/jemalloc`.

[6] Jemallocator Developers. *Jemallocator*. URL: `https://github.com/tikv/jemallocator`.

[7] Rust Developers. *Bindgen*. URL: `https://github.com/rust-lang/rust-bindgen`.

[8] UPPAAL Developers. *UDBM*. URL: `https://github.com/UPPAALModelChecker/UDBM`.

[9] Brook Heisler. *Criterion.rs*. URL: `https://github.com/bheisler/criterion.rs`.

[10] David Tolnay. *Paste*. URL: `https://crates.io/crates/paste`.