

Experimental Study of Compressed Stack Algorithms in Limited Memory Environments

Jean-François Baffier^{*1,2}, Yago Diez^{†3}, and Matias Korman^{‡3}

¹National Institute of Informatics

²JST-ERATO Kawarabayashi Large Graph project

³Tohoku University

June 16, 2017

Abstract

The *compressed stack* is a data structure designed by Barba *et al.* (Algorithmica 2015) that allows to reduce the amount of memory needed by an algorithm (at the cost of increasing its runtime). In this paper we introduce the first implementation of this data structure and make its source code publicly available.

Together with the implementation we analyze the performance of the compressed stack. In our synthetic experiments, considering different test scenarios and using data sizes ranging up to 2^{30} elements, we compare it with the classic (uncompressed) stack, both in terms of runtime and memory used.

Our experiments show that the compressed stack needs significantly less memory than the usual stack (this difference is significant for inputs containing 2000 or more elements). Overall, with a proper choice of parameters, we can save a significant amount of space (from two to four orders of magnitude) with a small increase in the runtime (2.32 times slower on average than the classic stack). These results holds even in test scenarios specifically designed to be challenging for the compressed stack.

Keywords— Stack algorithms, time-space trade-off, convex hull, implementation

1 Introduction

In recent years we have seen a huge growth in the everyday use of small devices such as smartphones, sensor networks, or even security cameras. These devices are becoming more and more powerful, but due to several constraints (ranging from budget issues to discouraging possible theft) it is sometimes desirable to keep them small in both memory size and computational power.

Consequently, theoretical computer science is expressing a renewed interest in the design of algorithms that use little space. A recent trend in the community has been the appearance of *time-space trade-off* algorithms [4]; that is, algorithms that can take into account space constraints;

^{*}J.-F. B. was supported by JST ERATO Grant Number JPMJER1305, Japan

[†]Y. D. was supported by the IMPACT Tough Robotics Challenge Project of Japan Science and Technology Agency

[‡]M. K. was supported in part by the ELC project (MEXT KAKENHI No. 12H00855, 15H02665, and 17K12635)

the larger the amount of memory that they have available, the less time that the algorithms will need. We refer the interested reader to [15] for a survey on the different models that have been proposed to handle space constraints.

From a theoretical point of view the interest has been in the relationship between time and space. In most cases, the dependency has been linear or almost linear [2, 3, 9, 16, 17]: that is, when we double the amount of available space we expect the runtime to more or less halve.

We believe that all these theoretical contributions are reaching a point where they can be used in practice. As a result, in this paper we take a more hands-on approach on the topic. Rather than studying theoretical dependences, we implement one of the proposed approaches to time-space trade-off and thoroughly assess its behaviour when executed using benchmark data of varying difficulty. Specifically, we are interested on seeing how much we can reduce the amount of memory consumed by algorithms while we make sure that runtimes remain reasonable.

Among the several results in this field, we focus on the *compressed stack* data structure introduced by Barba et al. [10]. The main reason, among several others, to choose this data structure for our analysis is that it is useful for several algorithms (as opposed to most techniques that are only be used for a specific problem); this data structure can be used to reduce the amount of space used by any deterministic incremental algorithm whose internal structure is a stack (see more details in Section 2).

Another good property of this structure is that, once properly implemented, the algorithm is unaware of which data structure it utilizes: it suffices to replace the classic (uncompressed) stack data structure with a compressed stack. This modular transparency makes it easy for users to adopt, and ideal for comparison purposes. Also, the dependency between time and space is not the same everywhere: for small amounts of memory the dependency is exponential (that is, by increasing the memory by a small constant we can *halve* the runtime), but it quickly becomes logarithmic afterwards (we need to *double* the amount of space to see any difference in the runtimes).

Thus, our study has two objectives. First, we want to verify that the theoretical dependency between time and space actually matches practice. Also, we want to provide some guidelines on how to find this breakpoint in the dependency so that the user can choose the right amount of memory to achieve the faster algorithm that fits their memory constraints.

1.1 Results and Paper Organization

Our main contribution is the implementation of the compressed stack data structure of Barba et al. [10]. The implementation is freely available at [6]. With the use of this library, one can implement any algorithm that uses this data structure quickly and efficiently (see examples of succinctness of the structure in subsection 3.6).

In Section 2 we give a brief overview of stack algorithms and the compressed stack data structure. In Section 3 we give a quick overview of how to use our library and discuss some minor differences between our implementation and the theoretical formulation by Barba *et al.*

In Section 4 we present a thorough study on the behaviour of the compressed stack in favourable and unfavourable scenarios (both constructed with synthetic data). We pay special attention to the comparison between the (theoretical) expected behaviour and the actual results obtained.

As expected, the compressed stack structure uses significantly less memory than the classic stack. From the theory we know that the running times must increase, but only a rough idea on the increase can be told. From the experiments done in this paper, we can deduce guidelines for prospective users so that the amount of memory is drastically reduced while we keep the runtimes relatively low. Further discussion about parameter settings is done in Section 5.

2 Preliminaries

The compressed stack data structure can only be used with a family of algorithms (called *stack algorithms*). This class includes widely used algorithms addressing problems such as computing the convex hull of a set of points, approximating a histogram by a unimodal function, or computing the visibility region of a point inside a polygonal domain. See [8, 10] for more examples of stack algorithms.

In full generality, we look at algorithms whose the input is a list of elements $\mathcal{I} = \{a_1, \dots, a_n\}$, and the goal is to find a subset of \mathcal{I} that satisfies a previously defined property. In a nutshell, we are looking at deterministic incremental algorithms that use a stack, and possibly other small data structures \mathcal{C} (this additional structure is called the *context* and ideally only consists in a few integers).

A stack algorithm solves the problem in an incremental fashion, scanning the elements of \mathcal{I} one by one. At any point during the execution, the stack keeps the values that form the solution up to that point. For each new element a that is taken from \mathcal{I} , the algorithm pops all values of the stack that do not satisfy a predefined "pop condition" and if a meets some other "push condition", it is pushed into the stack. The algorithm then proceeds to the next element in \mathcal{I} until all elements have been processed. The final result is normally contained in the stack, and at the end it is reported. This is done by simply reporting the top of the stack, popping the top vertex, and repeating until the stack is empty. Thus, an algorithm \mathcal{A} that follows this scheme is called a *stack algorithm* (see a pseudo-code in Algorithm 1).

ALGORITHM 1: Theoretical scheme of a stack algorithm

```
1 Initialize stack  $\mathcal{S}$  and context  $\mathcal{C}$  (auxiliary data structure) with  $O(1)$  elements from  $\mathcal{I}$ 
2 forall subsequent input  $a \in \mathcal{I}$  do
3   while  $\mathcal{A}.\text{popCondition}(a, \mathcal{C}, \mathcal{A}.\text{top}(1), \dots, \mathcal{A}.\text{top}(k))$  do
4      $\mathcal{A}.\text{pop}()$ 
5   end
6   if  $\mathcal{A}.\text{pushCondition}(a, \mathcal{C}, \mathcal{A}.\text{top}(1), \dots, \mathcal{A}.\text{top}(k))$  then
7      $\mathcal{A}.\text{push}(a)$ 
8   end
9 end
10  $\mathcal{A}.\text{report}()$ 
```

2.1 Sample problem: convex hull computation

A typical example of a stack algorithm is the convex hull problem: given a list of points p_1, \dots, p_n in the plane sorted in increasing values of their x -coordinate, we want to compute their convex hull, i.e., the smallest convex set that encloses all of the them. Among the many algorithms that solve this problem, [1]¹, the one by [18] falls in the class of stack algorithms.

Lee's algorithm processes the points sequentially and stores the elements that are currently candidates for being in the convex hull. For simplicity, we discuss how to compute the upper hull (i.e., points that are in the convex hull and are above the line passing through the rightmost and

¹The algorithms in this survey actually compute a slightly more general problem: computing the convex hull of a simple polygon, but both problems are almost identical. The simple polygon case has a few more difficult cases (such as when the polygon spirals around itself), but they have no impact on the way in which the stack is handled. Thus, for simplicity we only describe the simpler case.

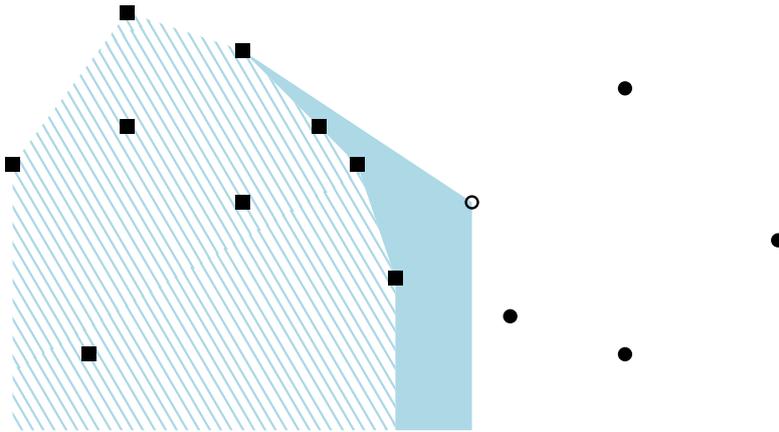


Figure 1: One step of Lee’s algorithm: the upper convex hull after processing 9 leftmost points (denoted by squares in the figure) and their convex hull is highlighted (dashed region). The next point (empty circle) is a witness that three points do not belong to the hull. These four points were the last ones to be added into the upper hull and thus they are popped from the stack. The updated hull is also shown (in solid).

leftmost point) of a set of points².

When a new element is processed it may be witness to several points that were previously in the upper hull and should not be there anymore (see Fig. 1). The key property is that those points must be the last ones that were considered as candidates. Consequently, they are removed in reverse order of insertion and thus a stack is the perfect data structure to store the list of candidates.

We refer the interested reader to [13] for more details on the convex hull problem and its applications. As an illustration on the simplicity in implementing stack algorithms, we have implemented this algorithm as part of our library (details are given in subsection 3.6).

2.2 Compressed Stack Overview

In this section we briefly describe the compressed stack data structure making a special emphasis on how can we save memory. During the execution of a stack algorithm, one may have many elements of the input in the stack. These elements are stored explicitly if we use the traditional stack resulting in high memory usage.

In the compressed stack structure, the user chooses a parameter p (to indicate the amount of space that the algorithm is allowed to use). Then the input is split into p blocks. Whenever a block has been fully processed (i.e., the incremental algorithm has scanned the last element of the block) we *compress* the block: rather than explicitly storing it, we store a small amount of information³.

This saves a lot of memory, since a block could have many elements in the stack but only a fixed amount of information per block is stored instead. Since we scan the input in a monotone

²The traditional algorithm scans once the input to compute the upper hull and a second time to compute the lower hull, but we note that the same algorithm can be modified to work in one pass by using two stacks. In any case, neither of these two options have a large impact on the overall working of the stack algorithm, so we ignore this and focus in the upper hull only.

³there exist two ways in which the block can be compressed, totally and partially. The basic concept is introduced in subsection 3.3. We refer the reader to Barba et al. [10] for complete details on this

fashion only one block can be partially processed at any instant of time. We store that block and its preceding block *explicitly* (i.e., most of the information is explicitly stored), while all other ones are somehow compressed.

Stack algorithms only need access to the top of the stack at any given time. Since this information is known by the compressed stack, we can perform as a usual stack. Eventually, the algorithm may pop many elements, and then the information inside a block that was compressed will be needed. This information can be *reconstructed* by re-executing the same algorithm, but only restricted to a portion of the input. The key trick to keeping the runtime small is to make sure that few reconstructions are needed, and always restricted to small portions of the input.

We emphasize that the working of the compressed stack is *transparent* to the algorithm. The algorithm is running, does some push and pop operations as well as reading the top of the stack. The stack data structure handles compression of information independently. Sometime during the execution, a pop will trigger a reconstruction. In this moment, the algorithm is paused and we launch a copy of the same algorithm (with a smaller input). Once the small execution ends, the needed information is available in memory, and we can resume with the main execution.

From a theory standpoint, stack algorithms run in $O(n^{1+\frac{1}{\log p}})$ time using $O(p \log_p n)$ space for any parameter $p \in \{2, \dots, n\}$. In particular, when p is a relatively large number (say, $p = 500$) the algorithm runs in slightly sublinear time, and uses logarithmic space. On the other hand, when $p = n^\epsilon$ the algorithm runs in linear time and uses $O(n^\epsilon)$ space. For comparison purposes, the usual stack runs in linear time and uses linear space, so the latter case all it consumes more memory without reducing the runtime (from a theoretical point of view).

3 Implementation

In this section we introduce our implementation of the stack algorithm framework. This section also aims at providing prospective users with practical information that enables them to add the compressed stack to their application or implement their own stack algorithms using the templates provided.

This library was implemented following the C++11 standard, and as such, requires a compiler that can support C++11⁴ [14]. As a convention, all the class members (fields or methods) starts with a ‘m’ followed by an upper case and all variables by a lower case. The entirety of pointers used within the library are instantiation of `shared_ptr`⁵. The code is available at [6] as an open source library under the MIT license. A preliminary but operational version — using Julia language [11] — was presented by Baffier et al. [5] and is available as [7] under the same MIT license.

The rest of this section is organized as follows: First, we describe the file organization of our library in subsection 3.1. The reading, treatment, and storage of the input data along with the description of the context structure follows in subsection 3.2. In subsection 3.3 we present the compressed stack data structure to be used within a stack algorithm. The whole library is then structured around the stack algorithm class as detailed in subsection 3.4. Additional functionalities are described in subsection 3.5. Finally, in subsection 3.6, we provide a couple of examples instantiating the stack algorithm class.

3.1 Class and file organization

Our stack algorithm library consists of three main classes: the `Data` class that handles the input, the `CompressedStack` class that instantiates transparently a space-optimized stack structure,

⁴That is one that accepts either `-std=c++11` or `-std=c++0x` compilation flags

⁵A smart pointer default class of the C++11 standard that automatically manages the memory of pointers sharing a common element. It follows the *resource acquisition is initialization* (RAII) programming idiom

and the `StackAlgorithm` class itself — respectively in subsections 3.2 to 3.4. These key classes, along with those they depend on, are all grouped in the `/include` folder as shown in Fig. 2.

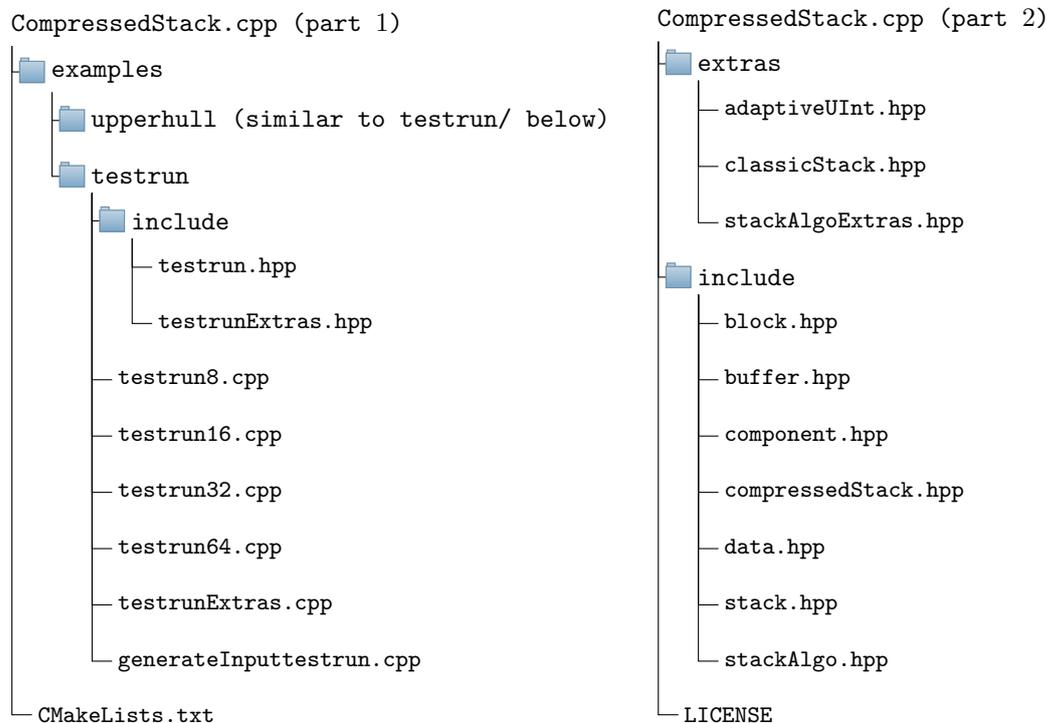


Figure 2: Directory tree of the `CompressedStack.cpp` library.

Extra functionalities — including an implementation of a classic stack and some measurement functions — are in a separate folder (named `/extras`). These classes are not needed for the execution of stack algorithms, but are useful for debugging and evaluation purposes. Finally, examples on how to implement stack algorithms (using normal or compressed stacks) can be found in the `/examples` folder.

3.2 Data, context and index types

In principle, the compressed stack algorithm can work with any kind of input data (depending on the application it could be numbers, points in the plane or edges of a large graph). In order to maintain the versatility, our implementation depends of several parameters provided by the users.

Because we use these abstract types, we cannot precompile the library. The abstract type is implemented in C++ via templates, causing the library to be header-only. The first set of parameters users have to fix relates to the type of input to be read (see Fig. 3). Specifically, the data type `D` needs to be provided so the algorithm is aware of the type of elements that it will receive from the input in the main loop of the stack algorithm.

Additionally, the user must fix the *index*. This is a type simply used to fix the maximum expected size of the input (so as to use a reasonable number of bits per input object). Samples for data index up to 8, 16, 32, and 64 bits unsigned integers are provided in the examples. For the sake of conciseness, the size of the index type is ignored for the rest of this article.

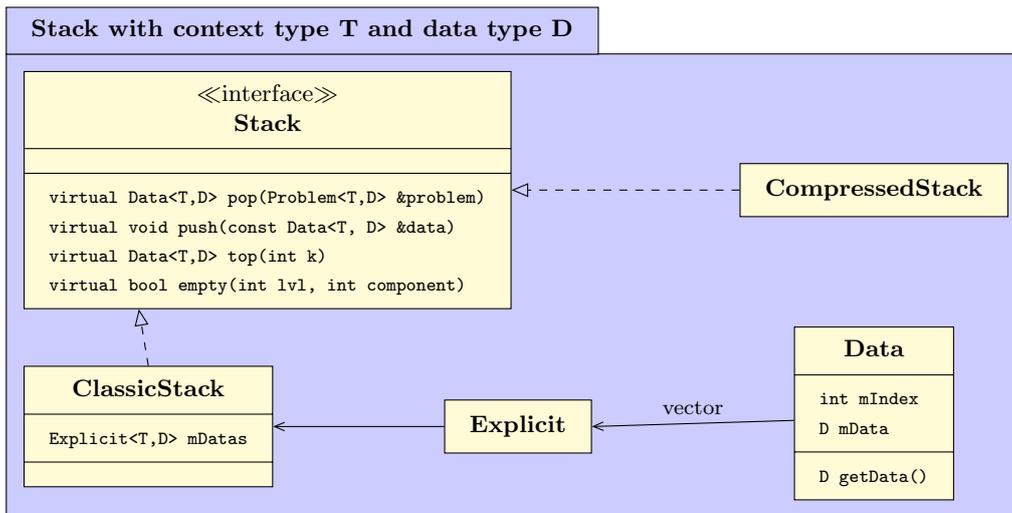


Figure 3: Class Diagram for the **Stack** interface and its implementation by **ClassicStack**. The **CompressedStack** implementation is described in Fig. 5. The namespace **std** is used implicitly for **vector**.

Finally, the user must fix the **context**. This context is a data structure of bounded size that is used during the execution of the stack algorithm. The space bottleneck of stack algorithms is the stack itself, but often they use a small amount of memory in their computations (for example, in the convex hull example, context is a bit used to prevent a degenerate case in which the convex hull spirals into itself). This context can be accessed (and modified) at any moment during the execution of the algorithm.

3.3 Compressed Stack

Our implementation pays special attention to modularity, so the stack algorithm is *transparent* to the kind of stack that is actually being used. That is, the algorithm sends push and pop requests and need not know if they are being handled by a regular or a compressed stack. All modifications needed to deal with space constraints are handled by the compressed stack class.

A stack, as shown by the **Stack** interface in Fig. 3, must provide a pop and a push function. It might also provide functions to check the top k element of the stack (for some small $k > 0$ given by the user). For convenience, a function to test the emptiness of a stack is also expected. Both the **CompressedStack** and the **ClassicStack** classes provided in this library instantiate this **Stack** interface.

Please see Fig. 4 for an overview of the working of the **CompressedStack** class. The key trick of saving space is to partition the input into blocks, that are recursively partitioned into blocks, and so on. Recall that a block may be stored *explicitly* (if it stores all elements of the block that have been pushed into the stack), *partially compressed* (in this case, we store information required to reconstruct portions of the block) or *fully compressed* (we store information required to reconstruct the block fully). This data structure compresses information that is unlikely to be accessed in the near future. Depending on the value of parameter p (set by the user) we may compress more or fewer blocks.

The resulting object-programming structure that constitutes the **CompressedStack** class is described in Fig. 5. The components store compressed data in their **mPartial** attribute. This

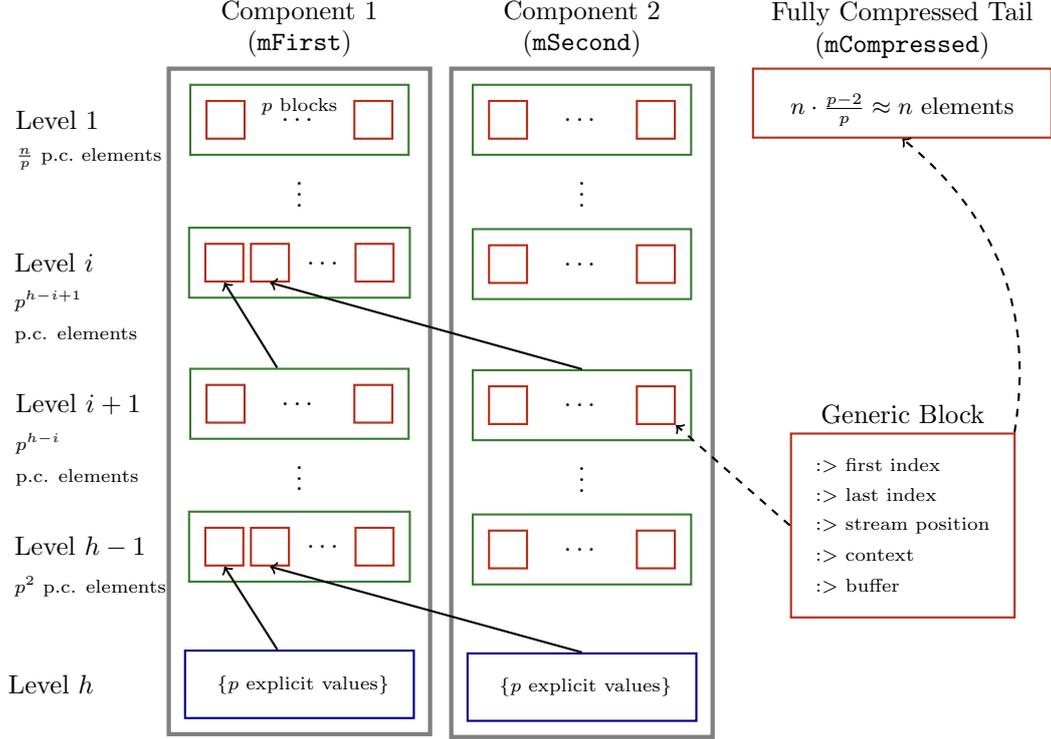


Figure 4: General sketch of a Compressed Stack: red boxes are blocks, green boxes are levels (vector of blocks), blue boxes are explicit values, plain arrows shows the partial compression (p.c.) of a level $i + 1$ into a block at level i . Recall that p is a parameter set by the user (denoting how much to compress the data), and that $h \approx \log_p n$ will denote in how many levels we subdivide the input.

takes the form of a matrix (vector of vectors) of blocks. Explicit data is stored in the `mExplicit` attribute. A *buffer* `mBuffer` stores the k top elements of a compressed stack that are required to be accessible by the user.

An object of the `CompressedStack` class includes the following attributes: The number of elements⁶ in the input \mathcal{I} as `mSize`. The space order p as `mSpace` and the depth h of each components as `mDepth`. A pointer to the current position in the stream, `mPosition` and a pointer to the current context `mContext`. The two necessary components of the stack that stores both compressed and explicit data: `mFirst` and `mSecond`. The fully compressed tail of elements `mCompressed`.

As described in Section 2, a compressed stack may need to self-reconstruct part of its compressed content that is stored in a *block*. This object is implemented as the `Block` class that stores the index of the first and last elements of the block. It also stores the position in the input stream, the context and an optional buffer.

⁶The case where the exact number is unknown is an additional functionality that is treated and explained in subsection 3.5. Although there might be some efficiency loss, the behaviour of the compressed stack is unchanged.

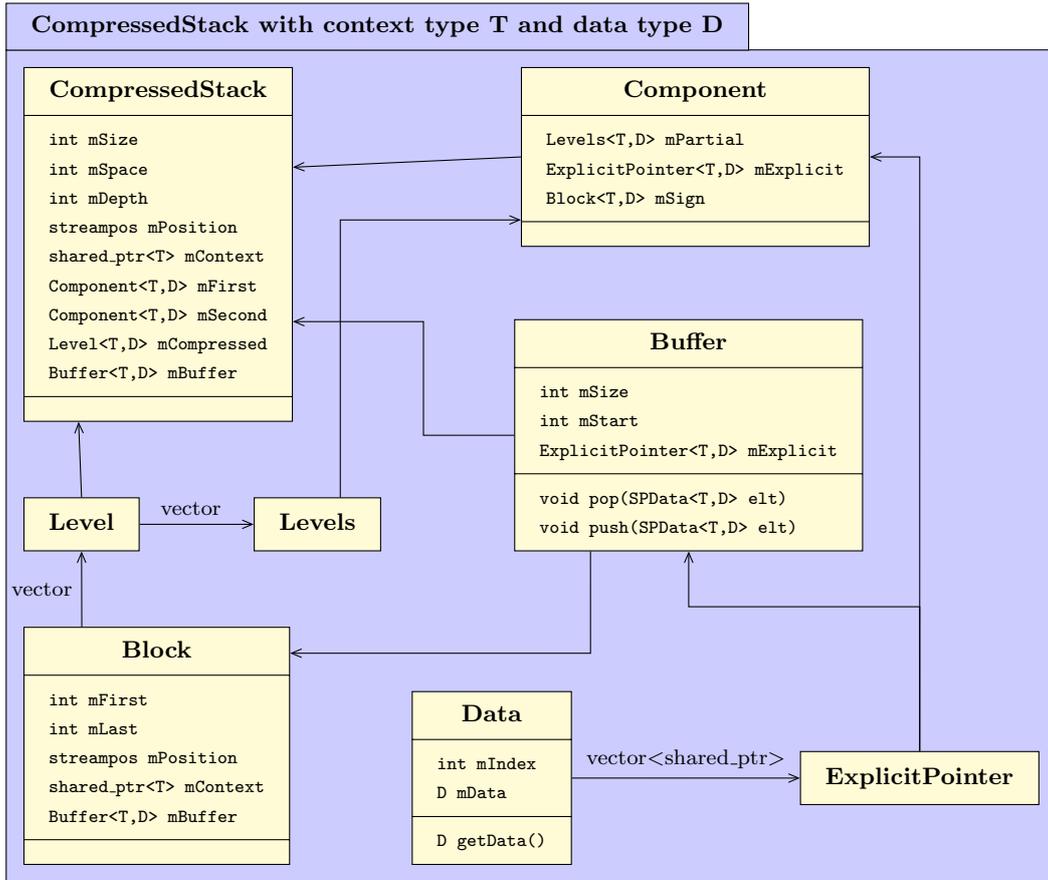


Figure 5: Class Diagram for the CompressedStack and related classes. The namespace `std` is used implicitly for `vector`, `shared_ptr`, and `streampos`.

3.4 Stack algorithm: Practical vs. Theoretical

In addition to the compressed stack data structure, we provide a framework for the compressed stack algorithms themselves (i.e., the general scheme described in Algorithm 1). Although their theoretical framework is complete and sound, we introduce some minor variations that may help in practical applications. See the modified version in Algorithm 2.

Consider a stack algorithm \mathcal{A} with associated stack \mathcal{S} (classic or compressed) and context \mathcal{C} . First, we describe the key features common to both Algorithms 1 and 2:

- **initialize**: initialize, if necessary, the empty stack \mathcal{S} and context \mathcal{C} . If the algorithm is allowed access to the top k elements of \mathcal{S} , then the execution of this function must provide at least k elements of \mathcal{I} into \mathcal{S} .
- **popCondition**: returns `true` if the top of the stack \mathcal{S} has to be popped.
- **pop**: pops the top element of \mathcal{S} .
- **pushCondition**: returns `true` if the last read element of \mathcal{I} should be pushed into the stack.
- **push**: pushes that element into \mathcal{S} .

ALGORITHM 2: Implementation of a stack algorithm

input: A stack algorithm \mathcal{A} with: an empty stack \mathcal{S} ; an empty context \mathcal{C} ; an input stream \mathcal{I} .

```
1  $\mathcal{A}$ .initialize()
2 while  $\mathcal{I}.EOF() == false$  do
3    $a \leftarrow \mathcal{I}.readInput()$ 
4   while  $\mathcal{S} \neq \emptyset$  do
5     if  $\mathcal{A}.popCondition(a)$  then
6        $\mathcal{A}.prePop(a)$ 
7        $a' \leftarrow \mathcal{S}.pop()$ 
8        $\mathcal{A}.postPop(a, a')$ 
9     else
10       $\mathcal{A}.noPop(a)$ 
11      break // exit the while loop
12    end
13  end
14  if  $\mathcal{A}.pushCondition(a)$  then
15     $\mathcal{A}.prePush(a)$ 
16     $\mathcal{S}.push(a)$ 
17     $\mathcal{A}.postPush(a)$ 
18  else
19     $\mathcal{A}.noPush(a)$ 
20  end
21 end
22  $\mathcal{A}.report()$ 
```

- **report:** \mathcal{A} execute a set of actions aimed towards reporting the objective of the algorithm once all the input data of \mathcal{I} has been processed. In most cases it explicitly returns the contents of the stack (report the top, pop from the stack, report the next one, and so on until the stack is empty).

Observe that the scheme of Algorithm 1 hides practical aspects, such as how to access the input values \mathcal{I} (on Line 2), or additional actions that could be executed together with either the push or pop (Lines 4 and 7). We provide these additional operations in Algorithm 2 defined as follows:

- **readInput:** operation executed each time an input element is read \mathcal{I} (e.g. if the input is provided in the form of an input file, this operation processes one line from the file and transforms it into data type D).
- **prePush** and **prePop:** set of actions to be executed before doing a push and pop, respectively.
- **postPush** and **postPop:** set of actions to be executed after a push and pop, respectively.
- **noPush** and **noPop:** set of actions to be executed when there is no push or pop, respectively.

The user can tune all the functions described above as needed. Other than **readInput**, by default the procedures do nothing (and can remain as such if not needed by the stack algorithm). Finally, the algorithms uses an End-Of-File operation **EOF**. This operation is used in Algorithm 2 at Line 2 and simply returns **true** once the end of input \mathcal{I} is reached (**false** otherwise).

All these functions and all basic stack functionalities are summarized in Fig. 6.

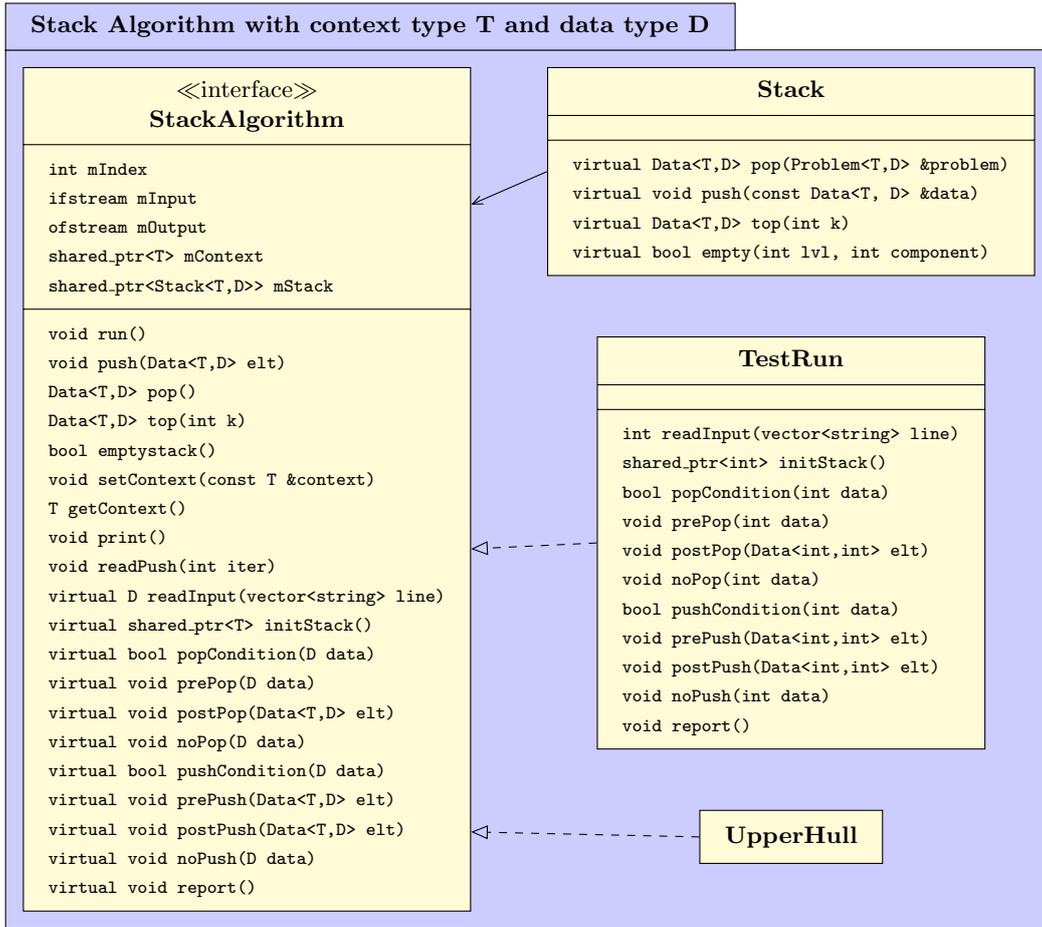


Figure 6: Class Diagram for the StackAlgorithm interface corresponding to Algorithm 2. The class TestRun is an implementation of StackAlgorithm for problem 1 where the context type $T = int$ and the data type $D = int$. The namespace `std` is used implicitly for `ifstream`, `ofstream`, `shared_ptr`, `vector`, and `string`.

3.5 Additional Functionalities

This section covers some additional functionalities of our library. Most of these functions aim to facilitate the evaluation and possible debugging of the library by the users. Documentation is available on [6].

Directly into the `StackAlgorithm` class we count the number of reconstructions executed during its execution. In addition, we provide functionalities to automatically handle an unknown input size (in a way that is transparent to the user). Ideally, the user should input a value n_{expect} that is somehow close to n . The compressed stack will work correctly regardless of the value given, but the more accurate the estimate is, the better the compressed stack will perform (if $n \ll n_{expect}$, the algorithm will be much slower, whereas $n_{expect} \ll n$ will cause too much memory to be used).

We also provide the option of using a classic stack (by itself or in parallel to the compressed stack). We also added a function to check that both the classic and the compressed stacks behave

similarly. The comparison between the behaviour of the two kind of stacks checks, after each `readInput`, `pop`, and `push` operation in the stack, that all information explicitly stored in the compressed stack matches the classic one.

We encourage the reader to add debugging and analysing tools within this extra framework and keep the original compressed stack class as light as possible. Contributions to [6] are welcome.

3.6 Example of Stack Algorithms

To illustrate the use of the compressed stack and of the stack algorithm, we provide two examples. This will also showcase how easy it is to implement new stack algorithms. The first example (problem 1) is a minimal one that can simulate any distribution of pops and pushes that we use in the experiments of Section 4. The upper hull problem introduced in subsection 2.1 is referred as problem 2. Both are fully implemented and available on [6]. We also provide an instance generator for both problems.

Problem 1 (Test Run). *This is an artificial stack algorithm that executes push and pop operations for debugging purposes. The data type D is a pair of positive integers separated by a comma. The first number indicates the value to be pushed into the stack whereas the second indicates the number of pops that should be done in lines 4-13 of Algorithm 2. The purpose of this algorithm will be clear in Section 4.*

Problem 2 (Upper Hull). *This algorithm computes the upper hull of a set of points in the plane, assuming that the input is given in increasing values of the x -coordinate. As shown in subsection 2.1, this can be done with a stack algorithm. For this problem, the data type D is a class `Point2D` that represents two dimensional points, and the input is simply the list of points as a pair of `float` coordinates. The output is an ordered list of points which represent the vertices of the upper hull.*

The algorithm of Lee does not need any context to compute the upper hull of a set of points that are sorted in the x -axis⁷, so it is simply set to a null pointer in C++11. Note that, during the execution of the algorithm, it will access the top two elements of the stack when determining whether or not to pop.

Since this algorithm fits in the stack algorithms class, it can be easily implemented (see a concise implementation in Fig. 7).

⁷If the points come sorted in a different way, then a context will be needed to prevent some degenerate cases. In any case, these degenerate cases have no impact on the usage of the stack, so we ignore them.

```

std::shared_ptr<emptyContext> initStack() {
    // first, read and push two values
    StackAlgo<emptyContext, Point2D, int>::readPush(2);
    // then initialize context (which in this case is NULL)
    std::shared_ptr<emptyContext> context;
    return context;
}

bool popCondition(Point2D last) {
    Point2D minus1, minus2;
    // read the two previous elements
    minus1 =
        StackAlgo<emptyContext, Point2D, int>::top(1).getData();
    if (StackAlgo<emptyContext, Point2D, int>::
        mStack->getBufferLength() < 2) {
        return true;
    }
    minus2 = StackAlgo<emptyContext, Point2D, int>::
        top(2).getData();
    // Pop condition is true depending on the points position
    if (Point2D::orientation(minus2, minus1, last) == 1) {
        return true;
    }
    return false;
}

bool pushCondition(Point2D data) {
    return true;
}

```

Figure 7: Instation of a `StackAlgorithm` template for the upper hull problem. Although we omit the code of the `Point2D::orientation` function, this implementation of the upper hull is concise, simple and independant of the kind of stack selected by the user.

4 Experimental Results

This section contains experimental studies aimed at analysing the practical performance of the compressed stack.

4.1 Data and evaluation measures.

In our experiments we use synthetic data. The main reason behind this decision is that it allows us to fully control the conditions in which the compressed stack would have to work (for example, we can control the number of reconstructions that will be executed). In addition, compressed stack algorithms spend a significant amount of time computing things that are unrelated to the stack (for example, in the upper hull algorithm, the `popCondition` operation must compute the determinant of a 3×3 matrix). Since these additional computations can affect the leading term for the running time, it make it hard to compare the performance of both stacks.

Our aim is to measure the differences between using a regular and a compressed stack. Thus, it is desirable to keep additional overhead to a minimum. As such, we implemented our experiments with the `testrun` problem described as problem 1 in Section 3.

The first experiment represents a very favourable situation for the compressed stack: a case in which data is accessed almost sequentially (and thus it is ideal for compression purposes). The goal in this case is to show the potential of memory saving that this data structure can achieve. The second test aims at setting a much more challenging scenario: continuous pops are set in a way such that scattered positions of the input file need to be accessed. This forces the compressed stack to repeatedly reconstruct portions of the input, and thus potentially lose a lot of time when compared to the classic stack.

In order to measure the performance of our implementation we focused on two magnitudes: the maximum amount of memory used by the algorithm and the running time. To measure the first, we used a heap profiler called *massif* [19] belonging to the Valgrind software suite. This software keeps track of the memory allocated in the execution heap at intervals of predefined length and outputs detailed heap memory usage. While this software allowed us to measure maximum memory usage, its use made the running of the algorithms much slower.

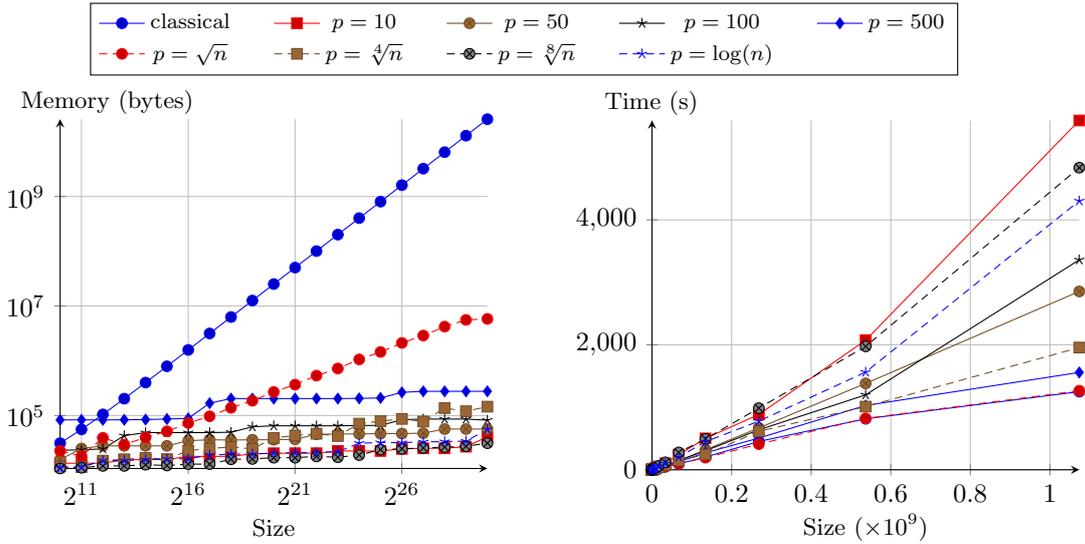
As the second magnitude that we were interested in was run-time, we needed to make two separate runs for every test case. In the first execution, we run *massif* alongside our code, obtaining memory usage data. In the second execution we run the test code alone in order to obtain unadulterated run-time readings. Consequently, in this section we present memory usage readings as outputted by *massif* (in bytes unless otherwise stated) as well as the times of the algorithms (in seconds) when run without *massif*. In order to be able to present values for widely different sizes, in each execution, we doubled the size of the input n (of size $n = 2^i$ for increasing values of i).

For comparison purposes, we also coded a classical stack class that implements the stack interface described in Fig. 3. This is simply a wrapper class for the widely used C++ `std::vector` that implements the `std::stack` interface. We believe that this is a representative instance of a classical stack using unconstrained memory. Regarding the parameters of the compressed stack, we focused on the ‘ p ’ space parameter introduced in subsection 2.2. For the purposes of this experiment, it suffices to know that the larger p is, the more space is used by the compressed stack (and fewer reconstructions are needed).

In order to illustrate the effect of this parameter in the performance of the compressed stack, we present results for eight different values of p . Specifically, the first four values are fixed (10, 50, 100 and 500) while the other four change with the size of the input n : \sqrt{n} , $\sqrt[3]{n}$, $\sqrt[4]{n}$ and $\log n$. Fixed values allow us to illustrate how an imbalance between n and p may result in very high running times for the compressed stack while the varying (and ever growing) values \sqrt{n} , $\sqrt[3]{n}$, $\sqrt[4]{n}$ and $\log n$ exemplify the trade-off between a lower memory limit for the compressed stack use and higher computation times. In order to keep the section within reasonable length limits, we only present summary figures of memory usage and running times. Detailed tables can be downloaded at [6].

4.2 Linear sized stack

In this first test we aim at creating a scenario that maximised the possibility of memory saving by the compressed stack with minor impact on the runtime. We consider the case in which the stack contains a linear fraction of the input. Specifically, fix a probability $\rho \in [0, 1]$; then every element of the input is pushed, and a pop will be executed with probability $1 - \rho$. In terms of the `testrun` problem defined, this stood for an input made up of a text file with a list of pairs of integers. For every pair, the first integer was a random positive number and the second was



(a) Memory comparison classical vs compressed. For ease of visualization, a logarithmic scale was used in both axes. (b) Time comparison classical vs compressed. No scaling in either axis is done for this figure.

Figure 8: As expected from theory, the normal stack uses a linear amount of space whereas the compressed stack only logarithmic. Regarding runtime, the classic stack has, as expected, the best performance of all. For large values of p (such as $p = \sqrt[4]{n}$) the running time is comparable to that of the classic stack. Conversely, for smaller values we can see that running times increase significantly. This is because smaller values of p need more reconstructions, which produce higher computation times.

chosen to be 1 with probability $1 - \rho$.

For any fixed $\rho > 0$ the expected size of the stack is ρn , and thus the memory used by the classic stack will be linear, whereas the compressed stack will only store a logarithmic amount of elements. Figures 8a and 8b show the memory used and runtime of our experiments for the case in which $\rho = 1$ (and thus no pops are ever executed). While we acknowledge that this is not a realistic situation, it does highlight the potential saving of space achieved by the compressed stack in cases where a large portion of it does not change.

To simulate more realistic situations, we also repeated the experiment with different values of ρ . In all cases, the tendencies observed were similar to the case without pops: the larger the value of ρ the fewer pops are executed, thus the more memory is needed (for example, the compressed stack with $p = \sqrt[4]{n}$ the memory used when $\rho = 1$, is between 1.6 and 2 times that of $\rho = 0.1$). However, the classic stack always needs a linear amount of space (for example the memory used increases around 24 times for the same parameters). On the other hand, with a larger number of pops, the compressed stack must be reconstructed more often, which brings noticeable increases in the running time (of 1.24 times slower on average on the case mentioned, for example). Since the overall performance for all values of ρ is similar, we refer the interested reader to [6].

Figure 8a depicts the maximum amount of memory needed in this test. A logarithmic scale (of base 2 for the x axis and base 10 for the y axis) is used for ease of visualization. The figure shows how in this test the classical stack needs much more memory than any of the compressed variants. There are two exceptional cases in which the classical stack uses less memory than a

compressed stack algorithm, but it only happens for extremely large values of p when compared to the input size (specifically, the compressed stack with $p = 500$ and input sizes of $2^{10}, 2^{11}$). This shows how the choice of parameter p is important to optimize the performance of the compressed stack. In this case p is too close to n ($n = 2^{10} = 1024$) and the structure of the stack is wasted as most values are kept explicitly.

The memory needed by the normal stack exceeds that of any compressed variant by two orders of magnitude already by size 2^{19} . This difference grows together with n (reaching four orders of magnitude for 2^{29}). The maximum memory needed by the classical stack in the test was over 25 Gigabytes for an input file of size 2^{30} (over 1000 million). Conversely, the compressed stack, in the worst case, only needed 54.6 Megabytes.

If we compare the different values of p for the compressed stack, we observe that, as expected, smaller values of p result in lower memory usage. Moreover, the results of algorithms with fixed values of p match the ones of variable p at expected values (for example, the algorithm with $p = 500$ should perform like the algorithm with $p = \sqrt{n}$ when $\sqrt{n} = 500 \leftarrow n = 250000 \approx 2^{19}$, and the two curves meet around that value). For fixed values of p it is easy to see that the memory grows logarithmically (specifically, we see a bump in the memory requirements when $\lceil \log_p n \rceil$ changes), whereas the growth is smoother for variable values of p . In all cases, the growth is very monotone, which matches the expectation from theory.

The downside of using a small value of p is that the running time will increase. The effect of the growth of p is only mildly visible in Figures 8a and 8b but will be more evident in subsection 4.3. The reason for this is that the number of times that the reconstruct function is invoked is small.

4.3 A challenging scenario for the compressed stack

In this section, we present a different test scenario that specifically tries to maximize the problems of the compressed stack: we set the input to produce push-pop cycles so as to force many reconstructions. Hence, overall the values that are pushed are at non-contiguous positions, making it difficult for the information to be compressed. Moreover, we make sure that the overall data that needs to be stored grows, but not at a linear rate. This is again a very artificial construction, but we believe that it shows that even under difficult conditions the compressed stack performs reasonably well. In order to create this setting, the instance forces the following operations into the stack:

- Push 8 elements, pop half of them (4).
- Repeat the previous step 8 times. At this point we have processed 64 elements and keep half of them (32) in the stack.
- Pop half of the stack, resulting in a stack of 16 elements.
- Repeat this double loop 8 times, resulting in 128 elements in the stack after 512 elements have been processed.
- We again pop half of the stack, keeping only 64 elements in the stack.
- Repeat now the triple loop 8 times, and so on

This procedure keeps adding cycles of increasing length until the desired input size n is reached. The stack stores 4 elements that are consecutive in the input, but the spacing between numbers to store grows exponentially, creating a very difficult situation for the compressed stack (since reconstruction operations will have to scan large portions of the input for just a few elements that are stored explicitly in the regular stack).

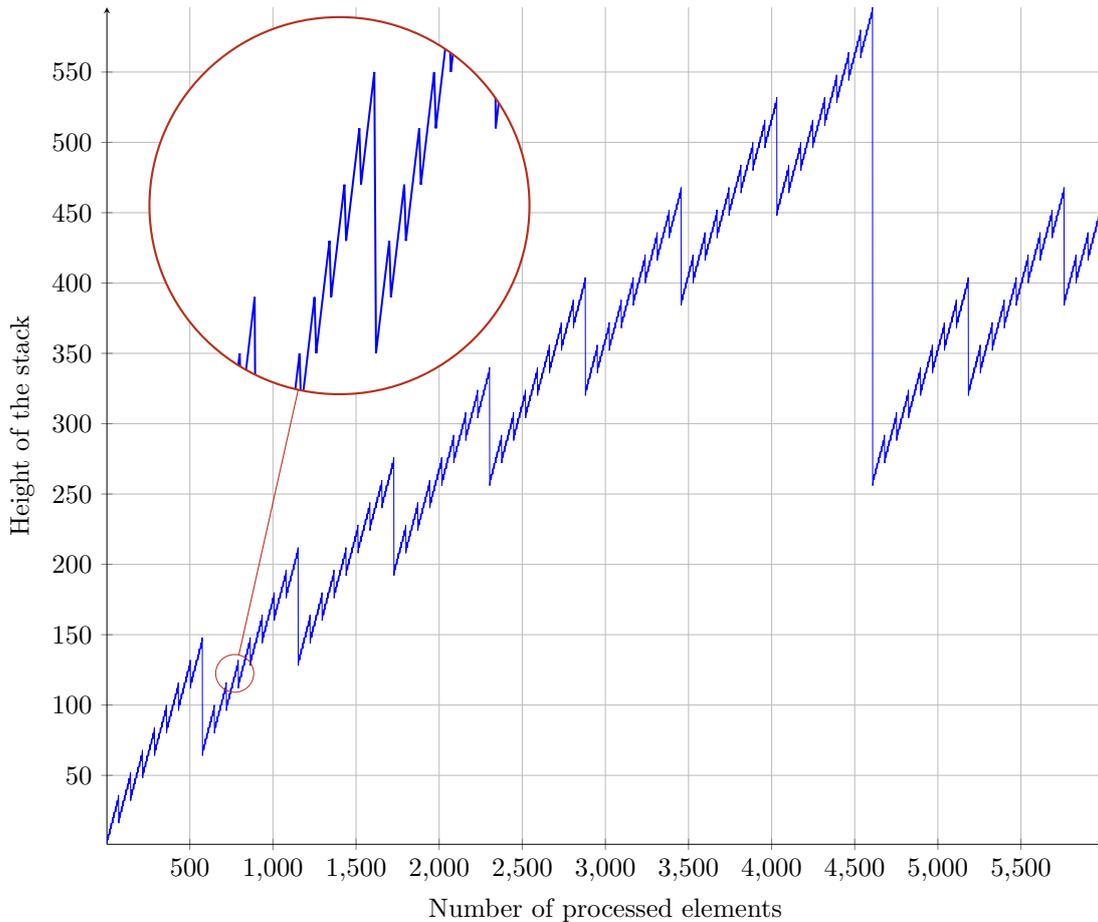
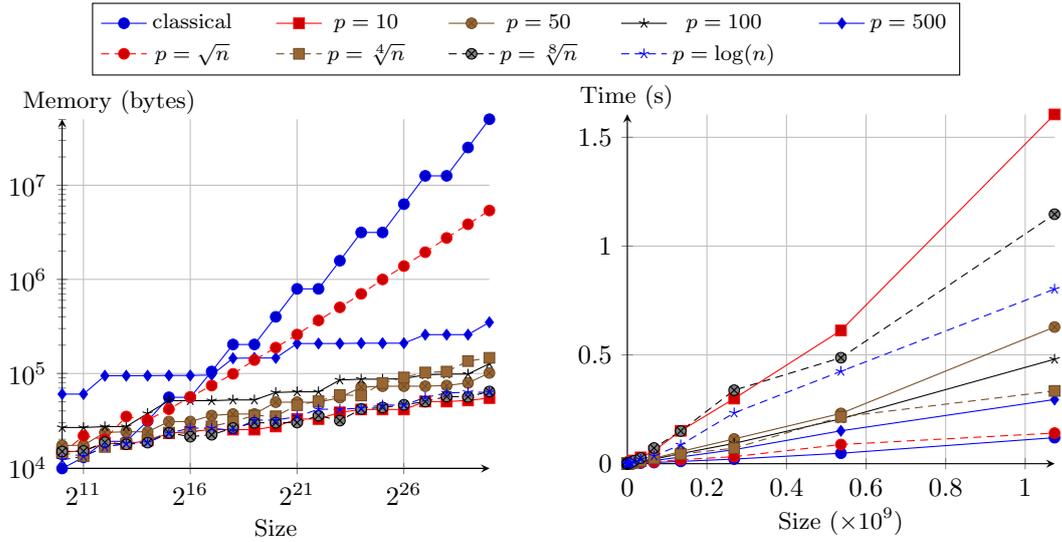


Figure 9: Number of elements in the compressed stack as a function of the number of processed elements. The zoomed image shows two levels of recursion, and outside the zoom we can see two more levels (for a total of four).

As in the previous example, n was taken following the powers of 2. Note that the choice of making loops with 8 iterations (and popping half of the stack at each step) is arbitrary. As 8 is a power of 2, the memory usage patterns in the classical stack become easier to predict (more details below). We call this test the *Christmas tree* test (because the height of the stack forms a Christmas tree-like shape, see Fig. 9 for a graphical representation on the number of elements present in the stack as a function of the input size).

Figure 10a shows the memory used in the Christmas tree test by all different stacks. We observe that the memory usage by the classical stack is significantly lower than in the previous experiment. This happens because very few elements are added into the stack (every factor of eight that the input grows, the space requirements only grow by a factor of 4, hence the stack has roughly $n^{2/3}$ elements).

This memory usage also grows stepwise in the sense that similar memory usages are detected for consecutive sizes. This is caused by the shape of the christmas tree (after a big pop like the one we find at around 4500 in Fig. 9 increases in the input size will not increase the amount of



(a) Memory comparison classical vs compressed (logarithmic scale used in both axes), CT test. (b) Time comparison classical vs compressed, CT test. Linear scale was used.

Figure 10: Christmas Tree experiments. In this test, designed to be challenging for the compressed stack, the memory saving respect to the classical stack is much smaller (and in a few instances the classical stack even needs less memory than some compressed stacks). Concerning time, the constant calls to the reconstruct function make the compressed stack much slower (as exemplified, for example by the behavior of the compressed stack with $p = 10$). However, even in this tailored scenario, we can see how the compressed stack maintains a capped memory usage as well as relatively low running times if the value of p is chosen appropriately (as can be seen for example, in the values of $p = \sqrt[4]{n}$).

memory needed until we reach a bigger loop).

We observe that the amount of memory needed by the compressed stacks is almost the same as the one needed in the previous experiment (the ratio varies slightly with every instance but stays between 0.8 and 1.2 of each other). The increase of the use of memory by the compressed stack in some cases is produced by the fact that the reconstruct function (frequently invoked in this example) duplicates parts of the input that might be of significant length.

It is important to notice that, although in this case, the memory saved by using the compressed stack is small (and in some of the cases with smaller n the compressed stack even needs more memory than the classical stack), the main reason for this is that the memory needed by the classical stack in this test is low. For example, the highest memory usage value in this test is four orders of magnitude lower than the maximum of the previous test. Even in such an ill conceived example, the maximum memory required by a compressed stack (with $p = \sqrt{n}$) is two orders of magnitude smaller than the classic stack. This shows how the memory used by the compressed stack stays capped even in the worse situations.

As expected, the number of reconstructions is much larger for the christmas tree instance. As such, the difference in the runtimes between the classic stack and a compressed one grows (as seen in Figure 10b). We observe that, although for small values of p the running times become unfeasible, for larger values the runtime is comparable to the one of a regular stack. For example, for $p = \sqrt[4]{n}$, the runtime in average increases by a factor of 2.32 (with 4.50 in the worst case).

This table exemplifies how the theoretical time-space trade-off is realised in practice and can itself constitute a starting point for prospective users of the compressed stack data structure. The left side of the table would be used to assess how much memory could be needed by every configuration of the compressed stack (expressed in the values of p) while the right side would provide an indication of the time penalty that the use of the compressed stack might produce. For example, in this case we believe the best compromise is obtained by the compressed stack with $p = \sqrt[4]{n}$ although the user should also chose the smaller value of p that fits their memory constraints in order to obtain the fastest running algorithm.

5 Conclusions

The experiments of this paper show that the compressed stack structure can be very useful not only from a theoretical point of view, but also on a practical level. Parallel to our work, a similar experimental study for another time-space trade-off problem previously only studied from a theoretical standpoint was done by [12]. Specifically, they studied the time-space dependency for the the problem of computing the shortest path between two points in a simple polygon. We believe that a trend of similar studies will soon follow for these or related problems.

Other than our preliminary implementation [7], this is the very first implementation of the compressed stack data structure introduced in [10]. The source code along with the data from the experiments presented in this paper is available for download as [6].

The results presented show how the compressed stack, along with other memory constrained algorithms has a huge potential to impact new technological contexts such as sensor networks or mobile phone apps. Specifically, Subsection 4.2 presented a (synthetic) situation where a normal stack needed 25 Gigabytes of memory while compressed stack implementations needed at most 55.5 Megabytes. This situation represents a challenge for current desktop computers and is infeasible in even the more advanced mobile phones (Apple’s Iphone 7, for example has 2 Gigabytes of RAM memory). Although this was a tailor-made case, it still shows how the property of the compressed stack of being able to limit memory usage by trading it for computation time has the potential of opening new application possibilities.

The reduction of memory of this data structure is undeniable, even in the very unfavourable scenario of the christmas tree. Moreover, the amount of memory needed in all scenarios is very stable (the amount of memory needed between the ideal and unfavorable scenarios lies between 0.9 and 1.2 of each other). This makes the compressed stack very robust at holding memory limitations, even for situations in which we do not know much about the structure of the input.

Although it was known that the saving of memory implies a runtime penalty, the exact amount was unclear. In this paper we have quantified how big much of a penalty to expect. The experiments show how the behaviour of the compressed stack can be predicted. Special attention has been given to comparing the practical behaviour to theoretical predictions. For example, the best value of p from a theoretical point of view is a fixed large constant as it gives the best time-space product. However, in our examples we have seen that a fixed value does not always perform well (for too small instances it may consume even more memory than the classic stack, and for larger instances the runtime may increase too much). Instead, values that depends on the size of the input (such as $\sqrt[4]{n}$) provide the best practical behaviour.

Each user of the compressed stack can choose the value of p so it fits the memory constraints in each specific situation and have an assessment on what the increase in run time will be. The examples presented in this paper can be used as guidelines by prospective users of the data structure to chose the values of p when they implement their applications using the stack algorithm templates provided in our library.

References

- [1] G. Aloupis. A history of linear-time convex hull algorithms for simple polygons, 2005. <http://cgm.cs.mcgill.ca/~athens/cs601/>.
- [2] B. Aronov, M. Korman, S. Pratt, A. van Renssen, and M. Roeloffzen. Time-space trade-offs for triangulating a simple polygon. *Journal of Computational Geometry*, 8(1):105 – 124, 2017.
- [3] T. Asano, K. Buchin, M. Buchin, M. Korman, W. Mulzer, G. Rote, and A. Schulz. Memory-constrained algorithms for simple polygons. *Computational Geometry: Theory and Applications*, 46(8):959–969, 2012. Special issue of selected papers from the 28th European Workshop on Computational Geometry.
- [4] T. Asano, K. Buchin, M. Buchin, M. Korman, W. Mulzer, G. Rote, and A. Schulz. Memory-constrained algorithms for simple polygons. *Computational Geometry: Theory and Applications*, 46(8):959–969, 2013.
- [5] J.-F. Baffier, Y. Diez, and M. Korman. Implementation of stack structure with limited memory. In *AA-AC*, 2016.
- [6] J.-F. Baffier, Y. Diez, and M. Korman. Compressed stack library (c++). <https://github.com/Azzaare/CompressedStacks.cpp.git>, 2016.
- [7] J.-F. Baffier, Y. Diez, and M. Korman. Compressed stack library (julia). <https://github.com/Azzaare/CompressedStacks.jl.git>, 2016.
- [8] N. Banerjee, S. Chakraborty, V. Raman, S. Roy, and S. Saurabh. Time-space tradeoffs for dynamic programming algorithms in trees and bounded treewidth graphs. In *COCOON*, pages 349–360, 2015.
- [9] L. Barba, M. Korman, S. Langerman, and R. I. Silveira. Computing the visibility polygon using few variables. *Computational Geometry: Theory and Applications*, 47(9):918–926, 2013.
- [10] L. Barba, M. Korman, S. Langerman, K. Sadakane, and R. I. Silveira. Space–time trade-offs for stack-based algorithms. *Algorithmica*, 72(4):1097–1129, 2014. ISSN 1432-0541. URL <http://dx.doi.org/10.1007/s00453-014-9893-5>.
- [11] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *CoRR*, abs/1411.1607, 2014.
- [12] J. Cleve and W. Mulzer. An experimental study of algorithms for geodesic shortest paths in the constant workspace model. In *EuroCG*, pages 165–168, 2017.
- [13] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008.
- [14] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, 2012. URL [http://www.iso.org/iso/catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372](http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372).
- [15] M. Korman. Memory-constrained algorithms. In *Encyclopedia of Algorithms*, pages 1260–1264. Springer, 2016.

- [16] M. Korman, W. Mulzer, A. van Renssen, M. Roeloffzen, P. Seiferth, and Y. Stein. Time-space trade-offs for triangulations and Voronoi diagrams. In *Algorithms and Data Structures Symposium (WADS)*, pages 482–494, 2015.
- [17] M. Korman, W. Mulzer, A. v. Renssen, M. Roeloffzen, P. Seiferth, and Y. Stein. Time-space trade-offs for triangulations and voronoi diagrams. *Computational Geometry: Theory and Applications*, 2017. Special issue of selected papers from the 31st European Workshop on Computational Geometry. In press.
- [18] D. T. Lee. On finding the convex hull of a simple polygon. *International Journal of Parallel Programming*, 12(2):87–98, 1983.
- [19] N. Nethercote, R. Walsh, and J. Fitzhardinge. Building workload characterization tools with valgrind. In *IISWC*, pages 2–2, Oct 2006.