Experimental Study of Compressed Stack Algorithms in Limited Memory Environments

Jean-François Baffier[1]

JSPS International Research Fellow — Department of Industrial Engineering and Economics, School of Engineering, Tokyo Institute of Technology, Tokyo, Japan
baffier.j.aa@m.titech.ac.jp

Yago Diez[2]

Yamagata University, Yamagata, Japan
yago@sci.kj.yamagata-u.ac.jp

Matias Korman[3]

Tohoku University, Sendai, Japan
mati@dais.is.tohoku.ac.jp

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 a certain class of algorithms 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 analyse 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 hold even in test scenarios specifically designed to be challenging for the compressed stack.

2012 ACM Subject Classification Theory of computation → Computational geometry

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

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.19

Related Version `https://arxiv.org/abs/1706.04708`, [8]

Supplement Material `https://github.com/Azzaare/CompressedStacks.cpp`

[1] Partially supported by JST ERATO Grant Number JPMJER1305, Japan.
[2] Partially supported by the Impact TRC project from Japan's Science and Technology agency.
[3] Partially supported by MEXT KAKENHI No. 12H00855, 15H02665, and 17K12635.

1 Introduction

Small computing devices have greatly increased their presence in our everyday lives in recent years. Most notably smartphones have become ubiquitous, but other devices such as sensors or security cameras have dramatically increased their number and everyday uses. These devices often have memory availability and computational power constraints due to price consideration, technology limitations or in order to discourage theft.

This growth tendency has coupled with the longstanding theoretical interest within the theoretical computer science community in the issue of memory usage in algorithmic design. This has yielded the creation of *time-space trade-off* algorithms [4]). These algorithms are designed so they work regardless of the amount of memory available but their performance improves with memory availability. We refer the interested reader to [15] for a survey on the different models that have been proposed to handle space constraints.

The major interest from a theoretical point of view is the relationship between time and space. In most cases, the dependency has been linear or almost linear [2, 3, 11, 16, 17]: that is, when we double the amount of available space we expect the runtime to more or less halve. However, several exceptions are known. For example, the best algorithm for the well-studied *minimum spanning tree* problem (given a set of points in the plane, embed a tree that spans all of them with the shortest possible length) needs cubic time to be solved in the presence of space constraints [5] (whereas it is solved in almost linear time when sufficient memory is available [13]).

On the positive side, for some problems we know that the dependency can be exponential. The focus of this paper is the *compressed stack* data structure introduced by Barba *et al.* [10]. This data structure applies to any deterministic incremental algorithm whose internal structure is a stack (more details below). When the space available is very small (say, s words of space for some s that is less than logarithmic in size when compared to the input), then the runtime is $O(n^2 \log n/2^s)$. Notice that the dependency is exponential: increasing the memory by a small constant we can *halve* the runtime. When the amount of space grows the dependency quickly becomes logarithmic (we need to *double* the amount of space to see any difference in the runtimes).

Another interesting property of the structure is that, if properly implemented, the algorithm is unaware of which data structure it utilizes: we only need to replace the classic (uncompressed) stack data structure with a compressed stack to obtain an algorithm that uses less memory. This modular transparency makes it very easy for users to adopt. More interestingly, it provides the ideal setting for comparison purposes.

We believe that this (theoretical) exponential dependency shows potential for usability in practical applications. In this paper we take a more hands-on approach on the topic. Rather than studying theoretical dependency, we implement the compressed stack and thoroughly assess its behavior when executed using benchmark data of varying difficulty. Specifically, we are interested in seeing how close the theoretical bound is to real runtimes, and if so what would be a good balance between time and space (that is, how much can we reduce the amount of memory consumed while making sure that runtimes remain reasonable). In order to tell the full story, it is also important to state that the use of the compressed stack is circumscribed to situations of high memory consumption. Thus those are the situations that we will consider throughout the paper. In those situations where the number of elements present in the stack is low enough not to ever represent a memory problem one should just use the classic stack.

Thus, our study has two objectives. First, 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 fastest 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]. This 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 as Problem 1 and in [8, 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 brief overview of our library. Due to lack of space, further description of the library as well as discussion of the minor differences between our implementation and the theoretical formulation by Barba *et al.* is available in [8, Section 3].

In Section 4 we present a thorough study on the behaviour of the compressed stack. Our study is based on two scenarios (one favourable and one unfavourable for the compressed stack). In our experiments we decided to use synthetic data. The main reason for this is that synthetic data allows us to focus only on the time and amount of memory needed by the stack data structure. Stack algorithms perform other computations apart from handling of the stack itself. For example, when computing the convex hull of a set of points we need to do triple orientation checks to decide if a point is popped from the stack (see Subsection 2.1 for a description of the problem and overview of the algorithm). These checks compute the determinant of a small matrix and when all of them are put together, they have a significant impact on the runtime. Although it is technically feasible to measure the computation time spent only in handling the stack for any given algorithm, this introduces precision errors and complicates the discussion of results. Furthermore using artificial data allows us to fully control the parameters in our experiments (input size, number of pushes and pops...). This allows us to focus on the properties that we want to study and give a clear view of the strong and weak points of the compressed stack.

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 the asymptotic trend can be also empirically observed. From the experiments done in this paper we can deduce more clear guidelines for prospective users so that they can drastically reduce the amount of memory consumed 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 certain 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 [9, 10] for more examples of stack algorithms.

In full generality, we look at algorithms whose input is a list of elements $\mathcal{I} = \{a_1, \ldots, a_n\}$, 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 $\mathcal{C}\rfloor\dashv\updownarrow$ (this additional structure is called the *context* and ideally only consists of 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 the pseudo-code in [8, Algorithm 1]).

2.1 Sample problem: convex hull computation

A typical example of a stack algorithm is the convex hull problem: given a list of points $p_1, \ldots, p_n$ 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 them. Among the many algorithms that solve this problem, [1][4], the one by Lee [18] falls in the class of stack algorithms.

The 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 leftmost point) of a set of points[5].

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 any more (see [8, Figure 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 [14] 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 [8, Subsection 3.6]).

2.2 Compressed Stack Overview

During the execution of a stack algorithm, one may have many elements of the input in the stack. In a classic stack these elements are stored explicitly, which may cause high memory requirements.

In the compressed stack structure we do not explicitly store all elements. First, the user chooses a parameter p to indicate the amount of space that the algorithm is allowed to use (the larger the p, the more memory it will 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 whatever part of it has been pushed into the stack, we store a small amount of information that can be used to afterwards determine exactly which elements were pushed.

---

[4] The 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.

[5] The 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.

Naturally, 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 fashion only one block is partially processed at any instant of time. We store that block and its preceding block in *uncompressed format* (i.e., almost explicitly), while all other ones are compressed[6].

Stack algorithms have access only to the top of the stack at any given time. Since the top element is part of an explicitly stored block, its information is always known and can be accessed as with 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, when the memory available is very small (i.e, we can only use s words of space for some $s \in o(\log n)$), stack algorithms run in $O(n^2 \log n/2^s)$ time. When the space available becomes $\Theta(\log n)$ the dependency in the runtime changes; the algorithm runs 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 superlinear time, and uses logarithmic space. On the other hand, when $p = n^\varepsilon$ the algorithm runs in linear time and uses $O(n^\varepsilon)$ space. For comparison purposes, the usual stack runs in linear time and uses linear space, so the latter case consumes more memory without reducing the runtime (this is of course from a theoretical point of view. Since additional reconstructions are needed the runtime will increase).

3 Implementation

In this section we give a brief overview of the `CompressedStacks.cpp` library. A complete description is given in [8, Section 3]. This library was implemented following the C++11 standard, and is available at [6] as an open source library under the MIT license.

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, and the `StackAlgorithm` class itself.

The `CompressedStack` implements the compressed stack data structure. As such, it handles push and pop operations (as well as all extra computations needed for compression). See Figure 1 for an overview of how data is stored in this class. The key trick of saving space is to partition the input into blocks, that are recursively partitioned into blocks, and so on. As introduced by Barba *et al.* [10], a block may be stored *explicitly* (if it stores all elements of the block that have been pushed into the stack), *partially compressed* (which allows reconstructing any portion of the block) or *fully compressed* (which allows fully reconstructing the block). This data structure compresses information that is further down

[6] There are two ways in which the block can be compressed, but this is is not relevant for the overview.

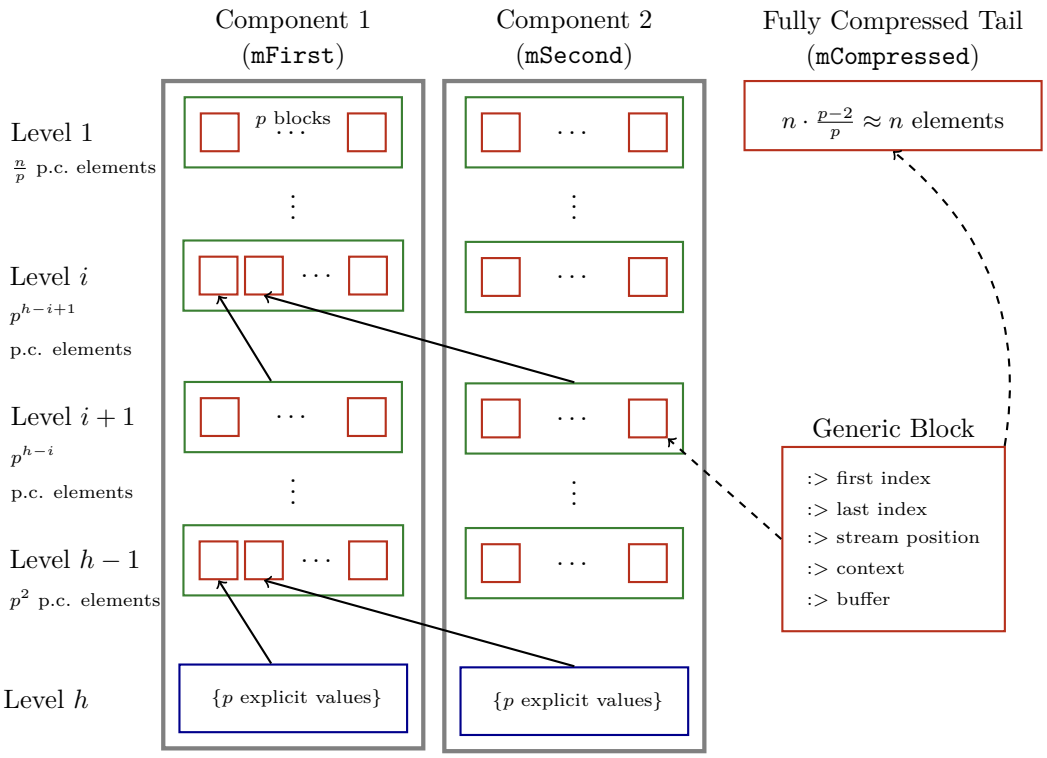


Figure 1 General sketch of a Compressed Stack: red boxes are blocks, green boxes are levels (vector of blocks), blue boxes are explicit values, plain arrows shows the partial compression (p.c.) of a level $i+1$ into a block at level i. Recall that p is a parameter set by the user (denoting how much to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

in the stack, thus unlikely to be accessed in the near future. Depending on the value of the parameter p (set by the user) we may compress more or fewer blocks. For comparison purposes we also provide an implementation of a classic stack that stores the input explicitly (and does not do any data compression).

The `StackAlgorithm` is a class used for solving the different stack algorithms. This is the class that the user needs to implement depending on his or her application. This class must be inherited and then a few operations like `pushCondition` or `popCondition` must be implemented (see more details in [8, Section 3]). The 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 (if used) are handled by the compressed stack class. Our library contains two examples of compressed stack algorithms. One is the upper hull problem (described in section 2.1) and the other is a synthetic problem described below.

▶ **Problem 1** (Test Run). *This is an artificial stack algorithm for experimental and debugging purposes. It reads the whole input and executes push and pop operations as requested. 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 after processing the first value (in lines 4-13 of [8, Algorithm 2]).*

4 Experimental Results

4.1 Data and evaluation measures

As mentioned in the introduction our experiments use synthetic data. Keeping in mind that our aim is to measure the differences between using a regular and a compressed stack we have tried to keep additional overhead to a minimum. In general, stack algorithms spend a significant amount of time in computations related to the problem being solved, but not related to the stack data structure being used. For example, in the upper hull algorithm, the `popCondition` operation must compute the determinant of a 3×3 matrix in order to determine whether a point is a candidate to belong to the upper hull. These computations can affect the leading term for the running time, obscuring the difference in performance of both stack types. Consequently, we present experiments using the testrun problem described as Problem 1 in Section 3 (an artificial problem with the smallest possible overhead).

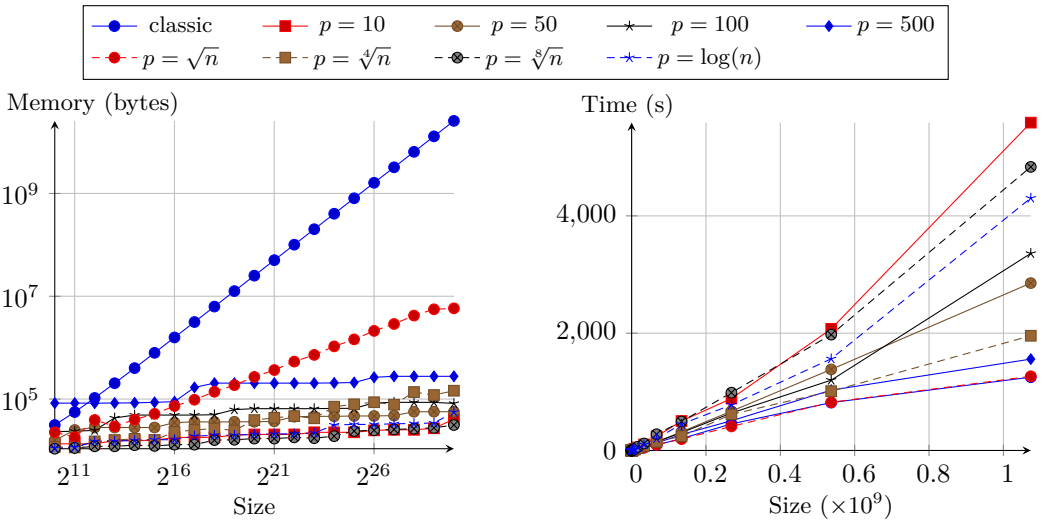
Another interesting property of synthetic data is that it gives us full control over the conditions in which both stacks have to work (for example, we can control the sparsity of the data and number of reconstructions that will be executed). We use this to our advantage and create two completely different scenarios in terms of the difficulty of data compression. 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 are 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.

For comparison purposes, all problem instances generated are solved with both the classic and compressed stack. In order to measure the performance we focused on two quantities: 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.

The second quantity 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 reported by massif (in bytes) 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).

The behaviour of the compressed stack highly depends on the 'p' 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: $\sqrt{n}, \sqrt[4]{n}, \sqrt[8]{n}$ and $\log n$. Fixed values allow us to illustrate how an imbalance between n and p may result in very high running times (or memory requirements). In order to obtain more



(a) Memory comparison classic vs compressed. For ease of visualization, a logarithmic scale was used in both axes.

(b) Time comparison classic vs compressed. No scaling in either axis is done for this figure.

Figure 2 As expected from theory, the classic stack uses a linear amount of space in a linear-space problem instance. In comparison, the compressed stack only uses a logarithmic space. Regarding runtime, the classic stack has, as expected, the best performance of all. However, for large values of p the running time is almost the same as of the classic stack (indeed, on average the compressed stack with $p = \sqrt{n}$ is only 1.031 times slower than the classic stack).

balanced values for all size ranges, we use varying values for p (as a function of n). 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, wiki section].

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. The first integer is the actual number that will be pushed into the stack (whose exact value is irrelevant and thus was a random integer) and the second was chosen to be equal to one with probability $1 - \rho$, or zero otherwise (recall that the second number is the number of pops to be executed).

After processing the whole input, the expected number of elements in the stack is ρn, and thus the memory used by the classic stack will be linear. Instead, the compressed stack will only store a logarithmic amount of elements (the exact number will depend on the value of the p parameter). Figures 2b and 2a 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. Moreover, in order to simulate more realistic situations we 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 $\rho = 0.1$).

Figure 2a 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 classic stack needs much more memory than any of the compressed variants. Note that, as expected, the memory consumed by the classic stack grows linearly with the input size.

There are two exceptional cases in which the classic 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 both cases p is too close to n ($n = 2^{10} = 1024$) and the structure of the stack is wasted as most values are kept explicitly nonetheless.

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 classic 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 5.8 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 the value of $\lceil \log_p n \rceil$ changes), whereas the growth is smoother for variable values of p. In both 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 2a and 2b 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, and the major time sink of the compressed stack is in the reconstruct operation. We defer a deeper analysis on runtime to the next experiment.

4.3 A challenging scenario for the compressed stack

We now consider a different test scenario that is tailored to be difficult for the compressed stack: we set the input to produce push-pop cycles in a way that forces many reconstructions. Moreover, the values that are pushed are at non-contiguous positions, making it difficult for the information to be compressed. We can also observe how 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).

We call this test the *Christmas tree* test (because the height of the stack forms a Christmas tree-like shape, see [8, Figure 9] for a graphical representation on the number of elements present in the stack as a function of the input size. As in the first scenario, we run Christmas tree instances whose total number of elements is a power of two. Note that the choice of making loops with 8 iterations (and popping half of the stack at each step) is arbitrary.

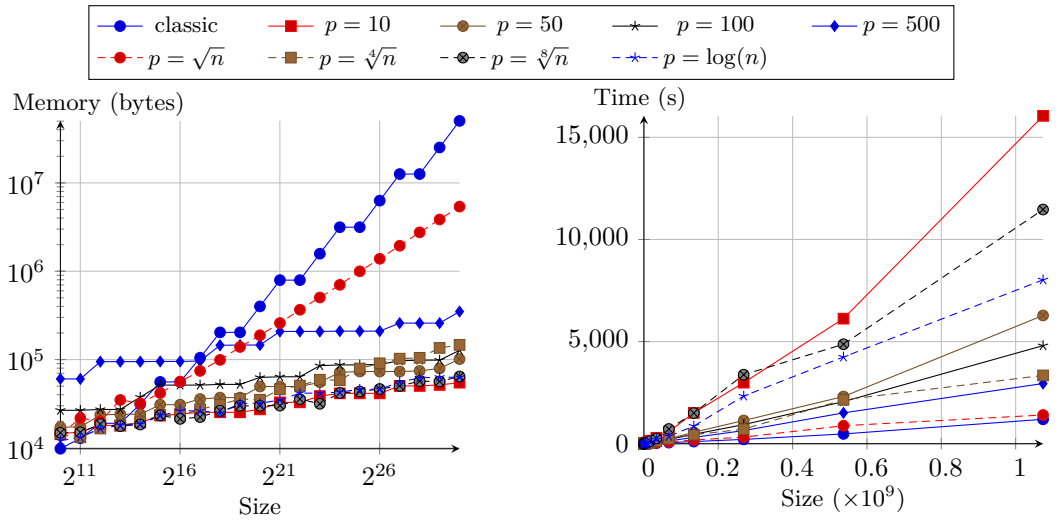
The main advantage of using a power of 2 for the number of iterations is that the memory usage patterns become easier to predict: imagine that for some value of n, the problem instance finished with a large pop removing half of the elements in its stack. When we double the instance size, the second half of the execution is spent in filling in the stack. Because the stack had been emptied, both executions are expected to use the same amount of memory.

Figure 3a shows the memory used in the Christmas tree test by all different stacks. Notice that the previously described stepwise pattern in memory usage is clearly visible. We also note that the amount of memory used by the classic stack is significantly lower than in the previous experiment. Again, this is because of significantly fewer 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.

We observe that at the amount of memory needed by the different parameters of the compressed stack is very similar to the one needed in the previous experiment. Indeed, for fixed values of n and p, the ratio of memory consumption in a linear-sized instance by the one needed by a Christmas-tree instance is very close to 1 (it depends on the exact value of n and p, but the average is 1.05 and variance is 0.1 or lower). This is consistent with the inner working of the compressed stack. Indeed, regardless of how many elements have been pushed, the compressed stack just stores a small amount of them explicitly (the exact amount will depend on n and p). The variation between the amount of memory needed by different instances happens because our implementation prioritizes saving as much memory as possible. Thus, if some block is empty we release that block of memory (and memory used by the smaller blocks nested inside).

The amount of memory saved by the compressed stack is significantly smaller than in the previous problem instance. This is to be expected, since fewer elements are pushed into the stack (and thus the classic stack needs less space). Indeed, for small values of n and extreme values of p the compressed stack even needs more memory than the classic stack. However, even in such an ill conceived example, when we use proper parameters of p is used (for example $p = \sqrt{n}$), the maximum memory required by a compressed stack is two orders of magnitude smaller than the classic stack.

The Christmas tree instance is specifically designed to force a lot of reconstructions. These reconstructions are computationally expensive, and as such we see a larger difference in runtimes between the classic stack and the compressed one (see Figure 3b). We observe that, although for small values of p the running times become infeasible, for larger values the runtime is comparable to the one of a regular stack. For example, for $p = \sqrt[4]{n}$, the runtime is in average 2.32 times slower than the classic stack (4.50 times slower in the worst case).



(a) Memory comparison classic vs compressed (logarithmic scale used in both axes), CT test.

(b) Time comparison classic vs compressed, CT test. Linear scale was used.

Figure 3 Christmas Tree experiments. In this test, designed to be challenging for the compressed stack, the memory saving in respect to the classic stack is much smaller (and in a few instances the classic 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 $p = \sqrt[4]{n}$).

4.4 Choosing the right value of p

From a theoretical point of view, setting p to be equal to a large constant gives the best performance for the compressed stack: a fixed value of p increases runtime over the classic stack by a logarithmic factor whereas space constraints are exponentially decreased. Values of p that depend on n have a worse time-space product.

Although this may be true in theory, in our experiments 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). We believe that in practical applications the value of p should depend on n. In this section we use the previous experiments to give guidelines on how to choose the correct value of p.

Naturally, the compressed stack is only recommended for cases in which the stack contains many elements (since otherwise a classic stack would be good enough). Now, assuming that the stack is going to be majorly full, there are two settings in which the compressed stack shines. In the first one we have a hard cap on the amount of memory available, and we need to make sure that the algorithms does not exceed that amount. In this case, we naturally want the largest possible value of p that keeps the memory requirements below the threshold.

A simple way to make sure this happens is to combine the compressed stack with a heap profiler. If at any point the memory constraints are exceeded, we stop and restart the algorithm with a smaller value of p. Although it will certainly work, such a brute force approach is not needed. In our experiments we observed that the memory used by the compressed stack is almost constant regardless of the scenario. We can use this to predict the amount of memory used without having to resort to a heap profiler. In our experiments,

we have observed that the amount of memory used is slightly below $300p\lceil \log_p n \rceil$ bytes. Of course, this value is not static and will depend on many parameters (such as operating system, compiler used, and so on).

More importantly, the exact constant will also depend in the type of elements that are pushed into the stack; For example, in our experiments we push integers. Our implementation of the compressed stack is abstract and can handle any data type, but the exact data type chosen will have an impact in the memory needed by the algorithms. In any case, given the stability of the space constraints of a compressed stack, it is not hard to predict the memory requirements of the compressed stack from a small sample of experiments.

Another scenario for the compressed stack to shine is when we want to overall keep memory requirements low, but no hard caps in the space requirements are present. In such cases, we believe that a practical compromise is obtained by fixing $p = \sqrt[4]{n}$: even in unfavourable scenarios it has a drastic reduction in memory consumption (orders of magnitude in difference as early as $n \approx 2^{15} \approx 32.000$) while it only brings a small increase in the runtime (in average 2.32 times slower than the classic stack).

5 Conclusions

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

The reduction of memory of this data structure is undeniable, even in the very unfavourable scenario of the Christmas tree. Specifically, Subsection 4.2 presented a (synthetic) situation where a normal stack needed 25 Gigabytes of memory while compressed stack implementations needed at most 5.8 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 showcases how the compressed stack nicely limits memory usage.

The drawback of the compressed stack is the increase in runtime when compared to a classic stack as known from its theoretical design. In this paper we have quantified how much of a penalty to expect as a function of p. This has allowed us to find a balance between the amount of memory saved and the increase in runtime. Most interestingly, we have observed that the amount of memory needed by the compressed stack is very stable regardless of the actual scenario. 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.

References

1 Greg Aloupis. A history of linear-time convex hull algorithms for simple polygons, 2005. URL: `http://cgm.cs.mcgill.ca/~athens/cs601/`.

2 Boris Aronov, Matias Korman, Simon Pratt, André van Renssen, and Marcel Roeloffzen. Time-space trade-offs for triangulating a simple polygon. *Journal of Computational Geometry*, 8(1):105–124, 2017.

3 Tetsuo Asano, Kevin Buchin, Maike Buchin, Matias Korman, Wolfgang Mulzer, Günter Rote, and André 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. doi:10.1016/j.comgeo.2013.04.005.

4 Tetsuo Asano, Kevin Buchin, Maike Buchin, Matias Korman, Wolfgang Mulzer, Günter Rote, and André Schulz. Memory-constrained algorithms for simple polygons. *Computational Geometry: Theory and Applications*, 46(8):959–969, 2013.

5 Tetsuo Asano, Wolfgang Mulzer, Günter Rote, and Yajun Wang. Constant-work-space algorithms for geometric problems. *Journal of Computational Geometry*, 2(1):46–68, 2011.

6 Jean-François Baffier, Yago Diez, and Matias Korman. Compressed stack library (C++). https://github.com/Azzaare/CompressedStacks.cpp.git, 2016.

7 Jean-François Baffier, Yago Diez, and Matias Korman. Compressed stack library (Julia). https://github.com/Azzaare/CompressedStacks.jl.git, 2016.

8 Jean-François Baffier, Yago Diez, and Matias Korman. Experimental study of compressed stack algorithms in limited memory environments. *CoRR*, abs/1706.04708, 2017. arXiv:1706.04708.

9 Niranka Banerjee, Sankardeep Chakraborty, Venkatesh Raman, Sasanka Roy, and Saket Saurabh. Time-space tradeoffs for dynamic programming algorithms in trees and bounded treewidth graphs. In *COCOON*, pages 349–360, 2015.

10 Luis Barba, Matias Korman, Stefan Langerman, Kunihiko Sadakane, and Rodrigo I. Silveira. Space–time trade-offs for stack-based algorithms. *Algorithmica*, 72(4):1097–1129, 2014. doi:10.1007/s00453-014-9893-5.

11 Luis Barba, Matias. Korman, Stefan Langerman, and Rodrigo. I. Silveira. Computing the visibility polygon using few variables. *Computational Geometry: Theory and Applications*, 47(9):918–926, 2013.

12 Jonas Cleve and Wolfgang Mulzer. An experimental study of algorithms for geodesic shortest paths in the constant workspace model. In *EuroCG*, pages 165–168, 2017.

13 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

14 Mark de Berg, Mark van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008.

15 Matias Korman. Memory-constrained algorithms. In *Encyclopedia of Algorithms*, pages 1260–1264. Springer, 2016.

16 Matias Korman, Wolfgang Mulzer, André van Renssen, Marcel Roeloffzen, Paul Seiferth, and Yannik Stein. Time-space trade-offs for triangulations and Voronoi diagrams. In *Algorithms and Data Structures Symposium (WADS)*, pages 482–494, 2015.

17 Matias Korman, Wolfgang Mulzer, André van Renssen, Marcel Roeloffzen, Paul Seiferth, and Yannik 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 Der-Tsai Lee. On finding the convex hull of a simple polygon. *International Journal of Parallel Programming*, 12(2):87–98, 1983. doi:10.1007/BF00993195.

19 Nicholas Nethercote, Robert Walsh, and Jeremy Fitzhardinge. Building workload characterization tools with valgrind. In *IISWC*, pages 2–2, Oct 2006. doi:10.1109/IISWC.2006.302723.