Space-Time Trade-offs for Stack-Based Algorithms

Luis Barba[1], Matias Korman*[2], Stefan Langerman[1], Rodrigo I. Silveira†[2], and Kunihiko Sadakane[3]

1 **Université Libre de Bruxelles (ULB), Brussels, Belgium.**
{lbarbafl,stefan.langerman}@ulb.ac.be
2 **Universitat Politècnica de Catalunya (UPC), Barcelona, Spain.**
{matias.korman,rodrigo.silveira}@upc.edu
3 **National Institute of Informatics (NII), Tokyo, Japan.**
sada@nii.ac.jp

Abstract

In memory-constrained algorithms we have read-only access to the input, and the number of additional variables is limited. In this paper we introduce the compressed stack technique, a method that allows to transform algorithms whose space bottleneck is a stack into memory-constrained algorithms. Given an algorithm $\mathcal{A}$ that runs in $O(n)$ time using a stack of length $\Theta(n)$, we can modify it so that it runs in $O(n^2/2^s)$ time using a workspace of $O(s)$ variables (for any $s \in o(\log n)$) or $O(n \log n/\log p)$ time using $O(p \log n/\log p)$ variables (for any $2 \leq p \leq n$). We also show how the technique can be applied to solve various geometric problems, namely computing the convex hull of a simple polygon, a triangulation of a monotone polygon, the shortest path between two points inside a monotone polygon, 1-dimensional pyramid approximation of a 1-dimensional vector, and the visibility profile of a point inside a simple polygon. Our approach exceeds or matches the best-known results for these problems in constant-workspace models (when they exist), and gives a trade-off between the size of the workspace and running time. To the best of our knowledge, this is the first general framework for obtaining memory-constrained algorithms.

1998 ACM Subject Classification I.3.5 Computational Geometry and Object Modeling

Keywords and phrases space-time trade-off, constant workspace, stack algorithms

Digital Object Identifier 10.4230/LIPIcs.STACS.2013.281

1 Introduction

The amount of resources available to computers is continuing to grow exponentially year after year. Many algorithms are nowadays developed with little or no regard to the amount of memory used. However, with the appearance of specialized devices, there has been a renewed interest in algorithms that use as little memory as possible.

Moreover, even if we can afford large amounts of memory, it might be preferable to limit the number of writing operations. For instance, writing into flash memory is a relatively

* M.K. was supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the European Union and was partially supported by the ESF EUROCORES programme EuroGIGA - ComPoSe IP04 - MICINN Project EUI-EURC-2011-4306.
† R.I.S. was supported by the FP7 Marie Curie Actions Individual Fellowship PIEF-GA-2009-251235 and was partially supported by the ESF EUROCORES programme EuroGIGA - ComPoSe IP04 - MICINN Project EUI-EURC-2011-4306.

30th Symposium on Theoretical Aspects of Computer Science (STACS'13).
Editors: Natacha Portier and Thomas Wilke; pp. 281–292

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

slow and costly operation, which also reduces the lifetime of the memory. Write-access to removable memory devices might also be limited for technical or security reasons. Whenever several concurrent algorithms are working on the same data, write operations also become problematic due to concurrency problems. A possible way to deal with these situations is considering algorithms that do not modify the input, and use as few variables as possible.

Several different memory-constrained models exist in the literature. In most of them the input is considered to be in some kind of read-only data structure. In addition to the input, the algorithm is allowed to use a small amount of variables to solve the problem. In this paper, we look for space-time trade-off algorithms; that is, we devise algorithms that are allowed to use up to s additional variables (for any parameter $s \leq n$). Naturally, our aim is that the running time of the algorithm decreases as s grows.

Many problems have been considered under this framework. In virtually all of the results, either an unconstrained algorithm is transformed to memory-constrained environments, or a new algorithm is created. Regardless of the type, the algorithm is usually an ad-hoc method tailored for the particular problem. In this paper, we take a different approach: we present a simple yet general approach to construct memory-constrained algorithms. Specifically, we present a method that transforms a class of algorithms whose space bottleneck is a stack into memory-constrained algorithms. In addition to being simple, our approach has the advantage of being able to work in a *black-box* fashion: provided that some simple requirements are met, our technique can be applied to any stack-based algorithm without knowing specifics details of their inner workings.

Stack Algorithms. One of the main algorithmic techniques in computational geometry is the incremental approach. At each step, a new element of the input is considered and some internal structure is updated in order to maintain a partial solution to the problem, which in the end will result in the final output. We here focus on *stack algorithms*, that is, incremental algorithms where the internal structure is a stack (and possibly $O(1)$ extra variables). A more precise definition is given in Section 2.

We show how to transform any such algorithm into an algorithm that works in memory-constrained environments. The main idea behind our approach is to avoid storing the stack explicitly, reconstructing it whenever needed. The running time of our approach depends on the size of the workspace. Specifically, it runs in $O(n \log n / \log p)$ time and uses $O(p \log n / \log p)$ variables (for any $2 \leq p \leq n$). In particular, when $p = n^{\varepsilon}$ the technique gives a linear-time algorithm that uses only $O(n^{\varepsilon}/\varepsilon)$ variables (for any $\varepsilon > 0$).

If only $o(\log n)$ space is available, we must restrict the class of algorithms considered slightly. We say that a stack algorithm is *green*[1] if, without using the stack, it is possible to reconstruct its top stack element in linear time (this will be formalized in Section 4.2). We show how to transform any green stack algorithm into one that runs in $O(n^2/2^s)$ time using $O(s)$ variables for any $s \in o(\log n)$.

Our techniques are conceptually very simple, and can be used with any (green) stack algorithm in an essentially black-box fashion. We only need to replace the stack data structure with the compressed data structure explained in Section 4.1, and create one or two additional operations for reconstructing elements in the stack. To the best of our knowledge, this is the first general framework for obtaining memory-constrained algorithms.

Applications. The technique is applicable, among others, to the following well-known and fundamental geometric problems (illustrated in Fig. 1). More details about these problems are presented in Section 5.

[1] or environmentally friendly.

Convex hull of a simple polygon The convex hull problem has already been studied in memory-constrained environments. Brönnimann and Chan [8] modified the method of Lee [17] so as to obtain several linear-time algorithms using memory-reduced workspaces. However, their model of computation allows in-place rearranging (and sometimes modifying) the vertices of the input and therefore does not fit in the memory constrained model considered here. In our model, the well-known *gift wrapping* or *Jarvis march* algorithm [15], reports the convex hull of a set of points (or a simple polygon) in $O(n\bar{h})$ time using $O(1)$ variables, where $\bar{h}$ is the number of vertices on the convex hull. In the same model, Chan and Chen [9] showed how to compute the upper hull of a sorted set of n points in linear time using $O(n^\varepsilon)$ extra variables for any fixed $\varepsilon > 0$.

Triangulation of a monotone polygon The memory-constrained version of this problem was studied by Asano *et al.* [3]. In that paper, the authors give an algorithm that triangulates *mountains* (a subclass of monotone polygons in which one of the chains is a segment). Combining this result with a trapezoidal decomposition, they give a method to triangulate a planar straight-line graph. Both operations run in quadratic-time in an $O(1)$-workspace.

Shortest path computation Without memory restrictions, the shortest path between two points in a simple polygon can be computed in $O(n)$ time [14]. Asano *et al.* [4] gave an $O(n^2)$ algorithm for solving this problem with $O(1)$-workspace, which later was extended to $O(s)$-workspaces [3]. Their algorithm starts with a (possibly quadratic) preprocessing phase that consists in repeatedly triangulating $\mathcal{P}$, and storing $O(s)$ edges that partition $\mathcal{P}$ into $O(s)$ subpieces of size $O(n/s)$ each. Once the polygon is triangulated, they compute the geodesic between the two points in $O(n^2/s)$ time by navigating through the sub-polygons. Our triangulation algorithm removes the preprocessing overhead of Asano *et al.* when restricted to monotone polygons.

Optimal 1-dimensional pyramid approximation Given an n-dimensional vector $f = (x_1, \ldots, x_n)$, find a unimodal vector $\phi = (y_1, \ldots, y_n)$ that minimizes the squared L_2-distance $||f - \phi||^2 = \sum_{i=1}^{n}(x_i - y_i)^2$. Linear-time algorithms for the problem exist [10], but up to now it had not been studied for memory-constrained settings.

Visibility polygon (or profile) of a point in a simple polygon Asano *et al.* [4] asked for a sub-quadratic algorithm for this problem in $O(1)$-workspaces. Barba *et al.* [6] provided a space-time trade-off algorithm that runs in $O(\frac{nr}{2^s} + n\log^2 r)$ time (or $O(\frac{nr}{2^s} + n\log r)$ randomized expected time) using $O(s)$ variables (where $s \in O(\log r)$, and r is the number of reflex vertices of $\mathcal{P}$). Parallel to this research, De *et al.* [11] proposed an $O(n)$ time algorithm using $O(\sqrt{n})$ variables.

We show in Section 5 that there exist green algorithms for all of the above applications (except for the shortest path computation), hence our technique results in new algorithms that run in $O(n^2/s)$ time for an $O(s)$-workspace (for $s \in o(\log n)$) or $O(n\log n/\log p)$ time using $O(p\log n/\log p)$ variables (for any $2 \leq p \leq n$). In particular, when $p = n^{1/\varepsilon}$, they run in linear-time using $O(n^\varepsilon)$ variables (for any constant $\varepsilon > 0$). The running time of the trade-off matches or exceeds the best known algorithms throughout its space range. Due to lack of space, many proofs have been deferred to the full version

2 Preliminaries

Given its importance, a significant amount of research has focused on memory-constrained algorithms, some of them dating back to the 1980s [20]. In this paper, we use a generalization of the constant-workspace model, introduced by Asano *et al.* [4, 5]. In this model, the input of the problem is in a read-only data structure. In addition to the input, an algorithm

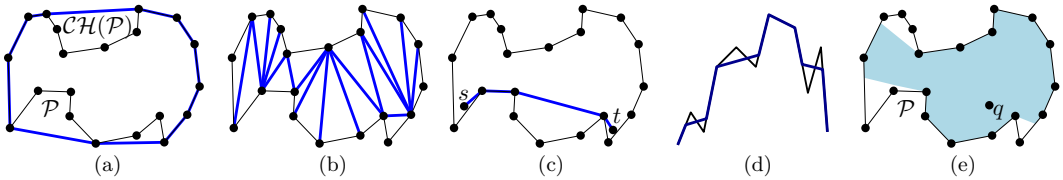

Figure 1 Applications of the compressed stack, from left to right: convex hull of a simple polygon, triangulation of a monotone polygon, shortest path computation between two points inside a monotone polygon, optimal 1-d pyramid approximation, and visibility polygon of a point $q \in \mathcal{P}$.

can only use a constant number of additional variables to compute the solution. Implicit storage consumption required by recursive calls is also considered part of the workspace. In complexity theory, the constant-work space model has been studied under the name of *log space* algorithms [2]. In this paper, we are interested in allowing more than a constant number of workspace variables. Therefore, we say that an algorithm is an s-workspace algorithm if it uses a total of $O(s)$ variables during its execution. We aim for an algorithm whose running time decreases as s grows, effectively obtaining a space-time trade-off. Since the size of the output can be larger than our allowed space s, the solution is not stored but reported in a write-only memory.

In the usual constant-workspace model, one is allowed to perform random access to any of the values of the input in constant time. The technique presented in this paper does not make use of such random access. Thus, unless the algorithm being adapted specifically needs it, our technique works in a more constrained model in which, given a pointer to a specific input value, we can either access it, or move the pointer to the previous or next input value. This is the case in which, for example, the input values are given in a doubly-linked list. We follow this model of computation and allow scanning the input as many times as necessary. Our model is particularly interesting when the input data cannot be modified, write operations are much more expensive than read operation, or whenever several programs need to access the same data concurrently.

Stack Algorithms

Let $\mathcal{A}$ be a deterministic algorithm that uses a stack, and possibly other data structures $\mathcal{DS}$ of total size $O(1)$. We assume that $\mathcal{A}$ uses a generalized stack structure that can access the last k elements that have been pushed into the stack (for some constant k). That is, in addition to the standard PUSH and POP operations, we can execute TOP(i) to obtain the i-th topmost element (for well-definedness purposes, this operation will return $\emptyset$ if either the stack does not have i elements or $i > k$).

We say that an algorithm is a *stack* algorithm if it follows the scheme in Algorithm 1. Notice that the scheme focuses on how the stack is handled, thus other operations could be present in $\mathcal{A}$, provided that the treatment of the stack is unaltered. For simplicity of exposition, we assume that only values of the input are pushed (see line 7 of Algorithm 1). In the general case one could push a tuple whose identifier is a. We allow this fact provided that the tuple has size $O(1)$. Essentially, our technique consists in replacing the $O(n)$-space stack of $\mathcal{A}$ by a *compressed stack* which uses less space. As we will see in Section 4.1, most of the time of our compressed stack structure is spent on computing the top element of the stack after a pop has been executed. For the case in which only $o(\log n)$ space is available, we must add one requirement to the algorithm. We require the existence of a GETTOP operation that, given an input value $a \in \mathcal{I}$ and a consecutive interval $\mathcal{I}' \subseteq \mathcal{I}$ of the input,

Algorithm 1 Basic scheme of a stack algorithm

```
1: Initialize stack and auxiliary data structure DS with O(1) elements from I
2: for all subsequent input a ∈ I do
3:    while some-condition(a,DS,STACK.TOP(1),..., STACK.TOP(k)) do
4:       STACK.POP()
5:    end while
6:    if another-condition(a,DS,STACK.TOP(1),..., STACK.TOP(k)) then
7:       STACK.PUSH(a)
8:    end if
9: end for
10: Report(STACK)
```

computes the $(k+1)$-th topmost element of the stack, provided that it belongs to $\mathcal{I}'$. This operation should run in $O(|\mathcal{I}'|)$ time using $O(s)$ variables. Whenever such operation exists, we say that $\mathcal{A}$ is *green*. Notice that this procedure need not be used by $\mathcal{A}$. In fact, in Section 5 we give a list of green algorithms, but none of them uses their corresponding GetTop operations. Full details on this operation are given in Section 4.2.

3 Compressed stack technique for $\Theta(\sqrt{n})$-workspaces

As a warm-up, we first show how to reduce the working space to $O(\sqrt{n})$ variables without increasing the asymptotic running time. Let $a_1, \ldots, a_n \in \mathcal{I}$ be the values of the input, in the order in which they are treated by $\mathcal{A}$. In order to avoid explicitly storing the stack, we virtually subdivide the values of $\mathcal{I}$ into blocks $B_1, \ldots, B_p$, such that each block B_i contains n/p consecutive values.[2] In this section we take $p = \sqrt{n}$. Then the size of each block will be $n/p = p = \sqrt{n}$. Note that, since we scan the values of $\mathcal{I}$ in order, we always know to which block the current value belongs to. Naturally, the stack can contain elements of different blocks. However, by the scheme of the algorithm, we know that all elements of one block will be consecutively pushed into the stack. We virtually group the elements in the stack according to the block that they belong to. At any point during the execution, we explicitly store the elements of the top two blocks in the stack. For the remaining blocks (if any), we store the first and last elements that were pushed into the stack. We say that these blocks are stored in *compressed* format.

For any input value a, we define the *context* of a as the content of the auxiliary data structure $\mathcal{DS}$ right after a has been treated. Note that the context occupies $O(1)$ space in total. For each block, regardless if we store it explicitly or in compressed format, we also store the context of the first element that was pushed into the stack.

It follows that for most blocks we only have the topmost and bottommost elements that we pushed into the stack (denoted a_t and a_b, respectively), but there could possibly be many more elements that we have not stored. For this reason, at some point during the execution of the algorithm we will need to reconstruct the missing elements in the compressed blocks of the stack. In order to do so we introduce a Reconstruct operation. Given a_t, a_b and the context of a_b, Reconstruct explicitly recreates all elements between a_b and a_t that existed in the stack right after a_t was processed.

[2] For simplicity of exposition, we assume that n is a power of p.

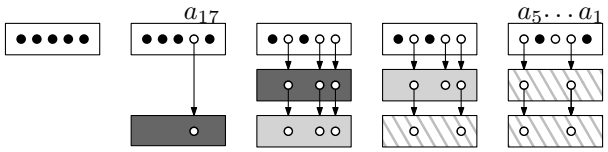

Figure 2 Push operation: the top row has the 25 input values partitioned into blocks of size 5 (white points indicate values that will be pushed during the execution of the algorithm; black points are those that will be discarded). The middle and bottom rows show the situation of the compressed stack before and after a_{17} has been pushed into the stack. Block $\mathcal{F}$ is depicted in dark gray, $\mathcal{S}$ in light gray, and the remaining compressed blocks with a diagonal stripe pattern.

▶ **Lemma 1.** RECONSTRUCT *runs in $O(m)$ time and uses $O(m)$ variables, where m is the number of elements in the input between a_b and a_t.*

Each time we invoke procedure RECONSTRUCT, we do so with the first and last elements that were pushed into the stack of the corresponding block. In particular, we have the context of a_b stored, hence we can correctly invoke the procedure. Also note that we have $m \leq n/p = p = \sqrt{n}$, hence this operation does not use any additional space. In order to obtain the desired running time, we must make sure that not too many unnecessary reconstructions are done. At any point of the execution, let $\mathcal{F}$ and $\mathcal{S}$ be the first and second topmost blocks in the stack, respectively. Recall that these are the only two blocks that are stored explicitly. Moreover, they are the latest blocks that we have visited and contained input values in the stack. There are two cases to consider whenever a value a is pushed: if a belongs to $\mathcal{F}$, it is added normally to the stack in constant time. Otherwise, we must create a new block containing only a. As a result, block $\mathcal{F}$ will become a new block only containing a, $\mathcal{S}$ will be the previous $\mathcal{F}$, and we must compress the former $\mathcal{S}$ (see Fig. 2). All these operations can be done in constant space by smartly reusing pointers. The pop operation is similar: as long as the current block $\mathcal{F}$ contains at least one element, the pop is executed as usual. If $\mathcal{F}$ is empty, we pop values from $\mathcal{S}$ instead. If after a pop operation the block $\mathcal{S}$ becomes empty, we pick the first compressed block from the stack (if any) and reconstruct it in full. Recall that we can do so in $O(\sqrt{n})$ time using $O(\sqrt{n})$ variables using Lemma 1. The reconstructed block becomes the new $\mathcal{S}$ (and $\mathcal{F}$ remains empty).

▶ **Theorem 2.** *The compressed stack technique can be used to transform $\mathcal{A}$ into an algorithm that runs in $O(n)$ time and uses $O(\sqrt{n})$ variables.*

Proof. The general workings of $\mathcal{A}$ remain unchanged, hence the difference in the running time (if any) will be due to push and pop operations. In most cases these operations only need a constant number of operations. The only situation in which an operation takes more than constant time is when pop is performed and block $\mathcal{S}$ becomes empty. In this situation, we must pay $O(\sqrt{n})$ time to reconstruct another block from the stack.

Recall that $\mathcal{A}$ scans the values of $\mathcal{I}$ in order: if at some point in the execution we push an element a belonging to a block B_i, we know that no element of block B_j (for some $j < i$) will afterwards be pushed into the stack. In particular, whenever the pop operation takes $O(\sqrt{n})$ time, we know that no element of the block associated to $\mathcal{S}$ is pushed (nor will be again be) into the stack. Thus, the cost of the reconstruction operation can be charged to that block. No block can be charged twice, hence at most $O(n/p)$ reconstructions are done. Since each reconstruction needs $O(p)$ time, the total time spent reconstructing blocks is bounded by $O(n)$. Regarding space use, at any point of the execution we keep at most two blocks in explicit form and the others compressed. The two top blocks need $O(p)$ space

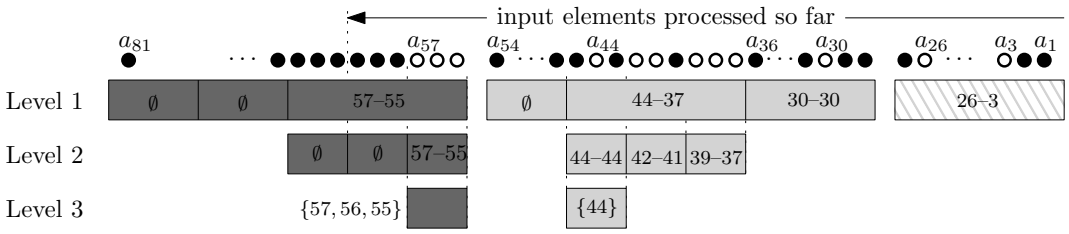

Figure 3 A compressed stack for $n = 81$, $p = 3$ (thus $h = 3$). The compression levels are depicted from top to bottom (for clarity, the blocks of the same level are not equally-sized). Color notation for points and blocks is as in Fig. 2. Compressed blocks contain the indices corresponding to the first and last element inside the block (or three pairs if the block is partially compressed); explicitly stored blocks contain a list of the pushed elements.

whereas the remaining at most $p-2$ blocks need $O(1)$ space each. Hence the space needed is $2(n/p) + p \times O(1)$, which equals $O(\sqrt{n})$ if $p = \sqrt{n}$. ◀

4 Compressed stack technique

In this section we present our compressed stack technique in full generality. For simplicity of exposition we describe it for the case in which $\mathcal{A}$ only accesses the topmost element of the stack (that is, $k = 1$). In the full version we explain how to extend the algorithm for larger values of k. We first present the technique for $\Omega(\log n)$-workspaces. Afterwards, we discuss the modifications needed for it to work on $o(\log n)$-workspaces.

4.1 For $\Omega(\log n)$-workspaces

In this section we generalize the above approach to the general case in which we partition the input into p blocks for any parameter $2 \leq p \leq n$ (the exact value of p will be determined by the user). Similarly to the previous case, we virtually decompose the input into p blocks of size n/p each. Instead of explicitly storing the top two non-empty blocks, we further subdivide them into into p sub-blocks. This process is then repeated for $h := \log_p n - 1$ levels until the last level, where the blocks are explicitly stored.[3]

We consider three different levels of compression: a block is either stored (i) *explicitly* if it is stored in full, (ii) *compressed* if only the first and last elements of the block are stored, or (iii) *partially-compressed*, if the block is subdivided into p smaller sub-blocks, and only the first and last element of each sub-block are stored. Analogously to the previous section, for each block in either compressed or semi-compressed format we store the context right after the first value of that block was pushed into the stack.

During the execution of the algorithm, the first level of compression will contain p blocks, with the top two partially compressed and the rest compressed. The i-th level of compression (for $1 < i < h$) will consist of two blocks of size n/p^{i-1} that are partially compressed. Thus each block is further subdivided into p sub-blocks of size n/p^i each. The first two non-empty sub-blocks are given to a lower level. In the lower level, the process continues recursively by further dividing the given sub-blocks. This process repeats until the h-th level, in which

[3] For simplicity in the explanation, we assume that our workspace is large enough to store the whole recursion. We note that this approach can be adapted for the case in which we have a workspace of $c \log n$ variables (for any $c > 0$). Details will be given in the full version of the paper.

the block size is $n/p^h = n/p^{\log_p n - 1} = p$, and is explicitly stored. Thus, in all but the lowest level, the top two blocks, denoted $\mathcal{F}_i$ and $\mathcal{S}_i$ for level i, are partially-compressed (whereas in the last level they are stored explicitly). See Fig. 3 for an illustration. Note that we allow blocks $\mathcal{F}_i$ to be empty, but blocks $\mathcal{S}_i$ can only be empty when the stack is empty.

▶ **Lemma 3.** *The compressed stack structure uses $O(p \log n / \log p)$ space.*

Proof. At the first level of the stack we have p blocks. The first two are partially-compressed and need $O(p)$ space each, whereas the remaining blocks are compressed, and need $O(1)$ space each. Since the topmost level can have at most p blocks, the total amount of space needed at the first level is bounded by $O(p)$.

At other levels of compression, we only keep two partially-compressed blocks (or two explicitly stored blocks for the lowest level). Regardless of the level in which it belongs to, each block needs $O(p)$ space. Since the total number of levels is h, the algorithm will never use more than $O(ph)$ space to store the compressed stack. ◀

Push operation. A push can be treated in each level $i \leq h$ independently. First notice that by the way in which values of $\mathcal{I}$ are pushed, the new value a either belongs to $\mathcal{F}_i$ or it is the first pushed element of a new block. In the first case, we register the change to $\mathcal{F}_i$ directly by updating the top value of the appropriate sub-block of $\mathcal{F}_i$ (or adding it to the list of elements if $i = h$). If the value to push does not belong to $\mathcal{F}_i$ we must create a new block, which will contain only a (and, if $i = 1$, we must compress the old $\mathcal{S}_i$). Since we are creating a block, we also store the context of a in the block. As in Section 3, these operations can be done in constant time for a single level.

Pop operation. This operation starts at the bottommost level h, and it is then transmitted to levels above. Naturally, we must first remove the top element of $\mathcal{F}_h$ (unless $\mathcal{F}_h = \emptyset$ in which case we must pop from $\mathcal{S}_h$ instead). In the simplest case, the block from which we popped has at least one more element. If this holds, no block is destroyed and the structure of the stack will not be affected. Note that we know which element of the stack will become the new top (since we have it explicitly stored). Thus, we need only transmit the new top of the stack to levels above. In those levels, we need only update the top element of the corresponding sub-block. The more complex situation happens when the pop operation emptied either $\mathcal{F}_h$ or $\mathcal{S}_h$. In the former case, we transmit the information to a level above. In this level we mark the sub-block as empty and, if this results in an empty block, we again transmit the information to a level above, and so on. During this procedure several blocks $\mathcal{F}_i$ may become empty, but no block of type $\mathcal{S}_j$ will do so (since $\mathcal{F}_i$ is always included in $\mathcal{F}_{i-1}$). Note that this is no problem, since in general we allow blocks $\mathcal{F}_i$ to be empty.

Finally, it remains to consider the case in which the pop operation results in block $\mathcal{S}_h$ becoming empty. First, we transmit this information to level above, which might also result in an empty block, and so on. We stop at a level i in which block $\mathcal{S}_i$ is empty (in which either $i = 1$ or $\mathcal{S}_{i-1}$ is not empty). We now must invoke the reconstruction procedure to obtain blocks $\mathcal{S}_j$ for all $j \geq i$. To reconstruct $\mathcal{S}_i$ we obtain from one level higher $(i-1)$ the first and last elements that correspond to the next non-empty sub-block after $\mathcal{S}_h$. Recall that at level $i-1$ we keep two blocks (of size n/p^{i-1}) in partially-compressed format. Hence, the values we need will be explicitly stored (since, by definition of i, block $\mathcal{S}_{i-1}$ will not empty after the pop is executed). If $i = 1$ and we reached the highest level, we pick the first compressed block and reconstruct that one instead. In either case, the first and last elements of the block to reconstruct are always known. Once $\mathcal{S}_i$ is reconstructed, we can proceed to reconstruct $\mathcal{S}_{i+1}$, and so on until we reconstruct $\mathcal{S}_h$.

Block Reconstruction. This operation is invoked when a block in the i-th level of compression needs to be reconstructed. We are given the first and last elements of that block that were pushed into the stack, denoted a_b and a_t, respectively, as well as the context right after a_b was inserted. Our aim is to obtain all stack elements between a_b and a_t right after a_t was pushed into the stack. This information should be in either explicit format (if $i = h$) or in partially-compressed format (if $i < h$). To reconstruct the block we use our algorithm recursively. The base case (i.e., if $i = h$) is handled with Lemma 1. For larger blocks, we execute $\mathcal{A}$ with the compressed data structure for a smaller size input (from a_b to a_t).

▶ **Lemma 4.** RECONSTRUCT *runs in* $O(m)$ *time and uses* $O(p \log m/\log p)$ *space, where* m *is the number of elements between* a_b *and* a_t.

▶ **Theorem 5.** *Any stack algorithm can be adapted so that, for any parameter* $2 \leq p \leq n$, *it solves the same problem in* $O(n \log n/\log p)$ *time using* $O(p \log n/\log p)$ *variables.*

4.2 For $o(\log n)$-workspaces

The previous technique can be used provided that the workspace is of size at least $\Omega(\log n)$. In the following we adapt it for smaller workspaces. From now on, we assume that $\mathcal{A}$ is green. The condition for an algorithm $\mathcal{A}$ to be green is to have a GETTOP operation. The general idea of this operation is the following: imagine that $\mathcal{A}$ is treating value a and at some point in time it pops the top element of the stack (denoted by t). Instead of reconstructing, we will invoke procedure GETTOP to find the new top element of the stack (denoted by ℓ). This operation must scan $\mathcal{I}$ until ℓ is found. Although we do not know exactly where ℓ lies, we will use the information in our compressed stack to guide this operation. Hence, for efficiency reasons, we restrict the procedure to look within a given interval. That is, procedure GETTOP receives three parameters: (i) the input value a that is generating the pop, (ii) two input values $t, b \in \mathcal{I}$, such that t is the current element at the top of the stack (i.e., the one that needs to be popped), and $b \neq t$ is another element that is in the stack, and (iii) the context information right before b was pushed into the stack. GETTOP must report the pair $(\ell, \text{CONTEXT}(\ell))$ in $O(m)$ time and $O(s)$ variables, where m is the number of input values between b and t in $\mathcal{I}$. Since b is in the stack and $b \neq t$, the value ℓ must always exist.

We apply the block partition strategy of the previous section, with $p = 2$ for s levels (recall that s is our allowed workspace). The only difference in the data structure occurs at the lowermost level, where each block has size $n/2^s$. Although we would like to store the blocks of the lowest level explicitly, the size of a single block is too large to fit into memory (if $s \in o(\log n)$, we have $n/2^s \in \omega(s)$). Instead, the blocks of the bottommost level are stored in compressed format. Recall that we store the context of the first element that is pushed into any block. Additionally, we store the context of STACK.TOP(1) (i.e. the top of the stack).

Push operations are handled exactly as in Section 4.1 (taking into account that the last level is now in compressed format): at each level it suffices to update the topmost element of $\mathcal{F}_i$, or create a new block containing the new input value (if it belongs to a new block).

Pop operations are also handled in a similar fashion as in Section 4.1. In most cases, we must remove one element from $\mathcal{F}_s$, unless the block is empty. In that case, we pop from $\mathcal{S}_s$. If both are empty, we pop from $\mathcal{F}_{s-1}$, and so on. This process ends when either $\mathcal{F}_1 \cup \mathcal{S}_1 = \emptyset$ (so the stack becomes empty after the pop) or we reach a block B_i of level i that does not become empty after the pop. As in Section 4.1 all blocks $\mathcal{S}_i$ of level $i < s$ that become empty must be reconstructed. This is done recursively using RECONSTRUCT. The only difference is at the bottommost level, where instead of reconstructing we use GETTOP with the top and bottom element of the block to obtain the new top element of the stack.

▶ **Lemma 6.** *The space used by a pop operation in $o(\log n)$-workspaces is $O(s)$. Moreover, the total time spent in all pop operations is $O(n^2/2^s)$.*

As in $\Omega(\log n)$-workspaces, the time bottleneck of the algorithm is given by the Pop operation, hence we obtain the following bounds.

▶ **Theorem 7.** *Any green stack algorithm can be adapted so that it solves the same problem in $O(n^2/2^s)$ time in an $O(s)$-workspace (for any $s \in o(\log n)$).*

5 Applications

In this section we show how our technique can be applied to several well-known geometric problems. For each problem we present an existing algorithm that is a (green) stack algorithm, where our technique can be applied to produce a space-time trade-off.

5.1 Convex hull of a simple polygon

Computing convex hulls is a fundamental problem in computational geometry, used as an intermediate step to solve many other geometric problems. For the particular case of a simple polygon $\mathcal{P}$ (Fig. 1(a)), there exist several algorithms in the literature that compute the convex hull of $\mathcal{P}$ in linear time (see the survey by Aloupis [1]). Among these, we highlight the one of Lee [17] that is green.

▶ **Lemma 8.** *Lee's algorithm for computing the convex hull of a simple polygon [17] is green.*

▶ **Theorem 9.** *The convex hull of a simple polygon can be reported in $O(n \log n / \log p)$ time using $O(p \log n / \log p)$ additional variables (for any parameter $2 \leq p \leq n$), or $O(n^2/2^s)$ time using $O(s)$ additional variables (for any parameter $s \in o(\log n)$).*

5.2 Triangulation of a monotone polygon

A simple polygon is called *monotone* with respect to a line ℓ if for any line ℓ' perpendicular to ℓ, the intersection of ℓ' and the polygon is connected. In our context, the goal is to report the diagonal edges of a triangulation of the given monotone polygon (see Fig. 1 (b)). Monotone polygons are a well-studied class of polygons because they are easier to handle than general polygons, can be used to model (polygonal) function graphs, and often can be used as stepping stones to solve problems on simple polygons (after subdividing them into monotone pieces). It is well-known that a monotone polygon can be triangulated in linear time using linear space [13].

▶ **Lemma 10.** *Garey* et al.*'s algorithm for triangulating a monotone polygon [13] is green.*

▶ **Theorem 11.** *A triangulation of a monotone polygon of n vertices can be reported in $O(n \log n / \log p)$ time using $O(p \log n / \log p)$ additional variables (for any parameter $2 \leq p \leq n$), or $O(n^2/2^s)$ time using $O(s)$ additional variables (for any parameter $s \in o(\log n)$).*

The only previous work on polygon triangulation in memory-constrained environment is due to Asano *et al.* [3]. In that paper, the authors give an algorithm that triangulates *mountains* (a subclass of monotone polygons in which one of the chains is a segment). Combining that result with a trapezoidal decomposition, they give a method to triangulate a planar straight-line graph. Both operations run in quadratic time in an $O(1)$-workspace. Our method speeds up the first half of their algorithm, hence if one were to obtain a time-space trade-off for computing the trapezoidal decomposition, we would instantly obtain a similar result for triangulating any polygon.

5.3 The shortest path between two points in a monotone polygon

Shortest path computation is another fundamental problem in computational geometry with many variations, especially queries restricted within a bounded region (see [18] for a survey). Given a polygon $\mathcal{P}$, and two points $p, q \in \mathcal{P}$, their *geodesic* is defined as the shortest path that connects p and q among all the paths that stay within $\mathcal{P}$ (Fig. 1(c)). It is easy to verify that, whenever $\mathcal{P}$ is a simple polygon, the geodesic always exists and is unique. The length of that path is called the *geodesic distance*.

Asano *et al.* [3,4] gave an $O(n^2/s)$ algorithm for solving this problem in $O(s)$-workspaces, provided that we allow an $O(n^2)$-time preprocessing. This preprocessing phase essentially consists in repeatedly triangulating $\mathcal{P}$, and storing $O(s)$ edges that partition $\mathcal{P}$ into $O(s)$ subpieces of size $O(n/s)$ each. Theorem 11 allows us to remove the preprocessing overhead of Asano *et al.* when $\mathcal{P}$ is a monotone polygon.

▶ **Theorem 12.** *Given a monotone polygon $\mathcal{P}$ of size n and points $p, q \in \mathcal{P}$, we can compute the geodesic that connects them in $O(n^2/s)$-time in an $O(s)$-workspace (for any $s \leq n$).*

5.4 Optimal 1-dimensional pyramid

A vector $\phi = (y_1, \ldots, y_n)$ is called *unimodal* if $y_1 \leq y_2 \leq \cdots y_k$ and $y_k \geq y_{k+1} \geq \cdots y_n$ for some $1 \leq k \leq n$. The 1-D optimal pyramid problem [10] is defined as follows. Given an n-dimensional vector $f = (x_1, \ldots, x_n)$, find a unimodal vector $\phi = (y_1, \ldots, y_n)$ that minimizes the squared L_2-distance $||f - \phi||^2 = \sum_{i=1}^{n}(x_i - y_i)^2$ (Fig. 1(d)). This problem has several applications in the fields of computer vision [7] and data mining [12,19]. Although the linear-time algorithm of Chun *et al.* [10] does not exactly fit into our scheme, it can be modified so that our approach can be used as well.

▶ **Theorem 13.** *The 1-D optimal pyramid for an n-dimensional vector can be computed in $O(n \log n/\log p)$ time using $O(p \log n/\log p)$ additional variables (for any parameter $2 \leq p \leq n$), or $O(n^2/2^s)$ time using $O(s)$ additional variables (for any $s \in o(\log n)$).*

5.5 Visibility profile in a simple polygon

In the visibility profile (or polygon) problem we are given a simple polygon $\mathcal{P}$, and a point $q \in \mathcal{P}$ from where the visibility profile needs to be computed. A point $p \in \mathcal{P}$ is visible (with respect to q) if and only if $pq \subset \mathcal{P}$, where pq denotes the segment connecting points p and q. The set of points visible from q is denoted by $\mathrm{Vis}_{\mathcal{P}}(q)$ and is called the *visibility profile* (or polygon) of q (see Fig. 1(e)). Visibility computations arise naturally in many areas, such as computer graphics and geographic information systems, and have been widely studied in computational geometry. Among several linear-time algorithms, we are interested in the method of Joe and Simpson [16] since it can be easily shown that it is green.

▶ **Lemma 14.** *Joe and Simpson's algorithm for computing the visibility profile [16] is green.*

▶ **Theorem 15.** *The visibility profile of a point q with respect to $\mathcal{P}$ can be reported in $O(n \log n/\log p)$ time using $O(p \log n/\log p)$ additional variables (for any $2 \leq p \leq n$), or $O(n^2/2^s)$ time using $O(s)$ additional variables (for any $s \in o(\log n)$).*

6 Conclusions

In this paper we have shown how to transform any stack algorithm so as to work in memory-constrained models, and presented several concrete applications where it can be applied.

Moreover, for many applications the technique can be applied in a black box fashion without altering the specifics of the algorithm. In addition, since the technique is rather simple to implement, we believe it can be useful in practice. A natural open problem is extending this approach to other data structures (e.g. trees), which would allow many other useful algorithms to work in memory-constrained workspaces.

References

1 G. Aloupis. A history of linear-time convex hull algorithms for simple polygons. http://cgm.cs.mcgill.ca/∼athens/cs601/.

2 S. Arora and B. Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

3 T. Asano, K. Buchin, M. Buchin, M. Korman, W. Mulzer, G. Rote, and A. Schulz. Memory-constrained algorithms for simple polygons. *CoRR*, abs/1112.5904, 2011.

4 T. Asano, W. Mulzer, G. Rote, and Y. Wang. Constant-work-space algorithms for geometric problems. *Journal of Computational Geometry*, 2(1):46–68, 2011.

5 T. Asano, W. Mulzer, and Y. Wang. Constant-work-space algorithms for shortest paths in trees and simple polygons. *J. Graph Algorithms Appl.*, 15(5):569–586, 2011.

6 L. Barba, M. Korman, S. Langerman, and R. I. Silveira. Computing the visibility polygon using few variables. In *ISAAC*, pages 70–79, 2011.

7 I. Bloch. Unifying quantitative, semi-quantitative and qualitative spatial relation knowledge representations using mathematical morphology. In *TFCV*, pages 153–164, 2003.

8 H. Brönnimann and T. M. Chan. Space-efficient algorithms for computing the convex hull of a simple polygonal line in linear time. *Computational Geometry: Theory and Applications*, 34(2):75–82, 2006.

9 T. M. Chan and E. Y. Chen. Multi-pass geometric algorithms. *Discrete & Computational Geometry*, 37(1):79–102, 2007.

10 J. Chun, K. Sadakane, and T. Tokuyama. Linear time algorithm for approximating a curve by a single-peaked curve. *Algorithmica*, 44(2):103–115, 2006.

11 M. De, A. Maheshwari, and S. C. Nandy. Space-efficient algorithms for visibility problems in simple polygon. *CoRR*, abs/1204.2634, 2012.

12 T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data mining with optimized two-dimensional association rules. *ACM Transactions on Database Systems*, 26(2):179–213, June 2001.

13 M. R. Garey, David S. Johnson, Franco P. Preparata, and Robert Endre Tarjan. Triangulating a simple polygon. *Information Processing Letters*, 7(4):175–179, 1978.

14 L. J. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. *Journal of Computer and System Sciences*, 39(2):126–152, October 1989.

15 R.A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18 – 21, 1973.

16 B. Joe and R. B. Simpson. Corrections to Lee's visibility polygon algorithm. *BIT Numerical Mathematics*, 27:458–473, 1987.

17 D. T. Lee. On finding the convex hull of a simple polygon. *International Journal of Parallel Programming*, 12(2):87–98, 1983.

18 J. S. B. Mitchell. Shortest paths and networks. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 607–642. 2nd edition, 2004.

19 Y. Morimoto, T. Fukuda, S. Morishita, and T. Tokuyama. Implementation and evaluation of decision trees with range and region splitting. *Constraints*, 2:401–427, 1997.

20 J. I. Munro and M. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12:315–323, 1980.