SERIE B — INFORMATIK

Lower Bounds for a Subexponential Optimization Algorithm

Jiří Matoušek*

B 92–15
July 1992

Abstract

Recently Sharir and Welzl [SW92] described a randomized algorithm for a certain class of optimization problems (including linear programming), and then a subexponential bound for the expected running time was established [MSW92]. We give an example of an (artificial) optimization problem fitting into the Sharir-Welzl framework for which the running time is close to the upper bound, thus showing that the analysis of the algorithm cannot be much improved without stronger assumptions on the optimization problem and/or modifying the algorithm. Further we describe results of computer simulations for a specific linear programming problem, which indicate that "one-permutation" and "move-to-front" variants of the Sharir-Welzl algorithm may sometimes perform much worse than the algorithm itself.

*Supported by Humboldt Research Fellowship. Department of Applied Mathematics Charles University Malostranské nám. 25, 118 00 Praha 1, Czechoslovakia, and Department of Computer Science, Free University Berlin Arnimallee 2–6, W-1000 Berlin 33, Germany

1 Introduction

Its is well-known that linear programming can be solved in time polynomial in the bit-size of the input ([Kha80]), but the number of arithmetic operations in all known polynomial-time algorithm depends on the input precision of the coefficients and it cannot be expressed as a function of d and n, the number of variables and constraints. A major open problem is to decide whether there exists a strongly polynomial algorithm for linear programming, which only needs a number of arithmetic operations polynomial in n and d. Until recently, the best bound was exponential. Then two randomized algorithms with a subexponential expected number of arithmetic operations, roughly $\exp(O(\sqrt{d}\log n))$, appeared: Kalai [Kal92] obtained one[1], and independently it was shown in [MSW92] that a randomized algorithm (resembling a dual simplex method) developed earlier by Sharir and Welzl [SW92] has a similar upper bound for the expected running time.

It is natural to ask whether the analysis of these algorithms can be still improved (perhaps to show that they are in fact polynomial), or whether the expected running time is close to the upper bounds in the worst case.

At present we cannot answer this question for linear programming itself. However, the analysis Sharir-Welzl algorithm is based on few simple properties of linear programming. The algorithm can in fact be formulated and analyzed in an abstract framework, which encompasses linear programming and various other optimization problems, satisfying few simple assumptions. In this paper we construct an artificial problem which fits into the abstract framework, and such that the expected running time of the Sharir-Welzl algorithm is close to the known upper bound. This shows that the analysis of the Sharir-Welzl algorithm for linear programming cannot be substantially improved without using more specific properties of linear programming and/or modifying the algorithm. We do not consider Kalai's algorithm here, but since it is similar to the Sharir-Welzl algorithm we expect that similar results can be obtained for it (one of Kalai's variants is in some sense dual to the Sharir-Welzl algorithm).

The Sharir-Welzl algorithm is formulated as a recursive procedure. For implementation, it would be simpler to have an iterative variant, and also to restrict the amount of randomness needed. For a linear programming algorithm due to Seidel [Sei91] (a predecessor of the Sharir-Welzl algorithm), whose basic formulation is also recursive, one can do this very elegantly: Welzl [Wel91] describes a modification which first puts the constraints into a random order (i.e., generates one random permutation), and then proceeds entirely deterministically, iterating a simple step. The expected running time is shown to be the same as for the basic version [Wel91].

Formulating such a one-permutation variant for the Sharir-Welzl algorithm is not difficult and the resulting algorithm is very easy to implement. A very similar algorithm was proposed by Megiddo [Meg92]; this algorithm is formulated in a dual setting (as a variant of Kalai's algorithm) and it is in fact the simplex algorithm with Bland's least index pivoting rule, where one puts the columns into a random order before the algorithm starts.

However, the analysis used for the Sharir-Welzl algorithm breaks down for the one-permutation variant, and to our knowledge no subexponential upper bounds for the expected running time are known. We provide an example (this time a linear programming problem,

[1]In fact, he gives several variants of the algorithm with small differences in the running time bounds; they are all more or less complicated randomized variants of the simplex algorithm.

with d variables and $2d$ constraints) indicating that there indeed is a substantial difference in the behavior of the one-permutation variant: while the Sharir-Welzl algorithm with any initial basis has expected running time $O(d^2)$, for the one-permutation variant computer experiments indicate that the expected running time is much larger, perhaps about $e^{\sqrt{d}}$.

In [Wel91], Welzl proposes still another version of Seidel's algorithm, which he calls "move-to-front". Here the algorithm again starts with a random order of the constraints, but then it keeps changing it (deterministically), trying to put constraints which were in some sense important for the computation so far to the beginning (hence the name). The hope is that this makes the algorithm converge much more quickly to the solution, and indeed computer experiments have shown that this strategy speeds up Seidel's algorithm significantly, at least for the examples used in the experiments. Analogously one can formulate a move-to-front version of the Sharir-Welzl algorithm. Here it is not entirely obvious what the correct analog should be; we have selected the one which seemed most natural. The experiments show that for our particular linear programming example, this variant is still much slower that the one-permutation variant. It is fair to say that the move-to-front strategy is intended for the situation where the number of constraints is much larger than the number of variables, while our example has $2d$ constraints and d variables, so it is no reason for discarding the strategy as a bad or useless one. Rather, it should serve as an example that a seemingly innocent heuristic modification to a randomized algorithm, destroying the assumptions on which the analysis was based, can make the performance quite poor.

2 The Sharir-Welzl algorithm

We begin by recalling the abstract framework of [SW92] (with minor formal modifications). In this setting, an optimization problem is a pair (H, w), where H is a finite set, and $w : 2^H \rightarrow \mathcal{W}$ is a function with values in a linearly ordered set $(\mathcal{W}, \leq)$. The elements of H are called the *constraints*, and for a subset $G \subseteq H$, $w(G)$ is called the *value* of G.

A *basis* B is a set of constraints with $w(B') < w(B)$ for all proper subsets B' of B. A *basis for* a subset G of H is a basis B with $B \subseteq G$ and $w(B) = w(G)$. So a basis for G is a minimal subset of G with the same value as G. The goal is to find a basis for H (and its value).

We say that a constraint $h \in H$ *violates* a basis B, if $w(B \cup \{h\}) > w(B)$. The maximum cardinality of any basis is called the *combinatorial dimension of* (H, w) and denoted by $\dim(H, w)$.

Let $w_0 \in \mathcal{W}$ be a value. The problem (H, w) is called an *LP-type problem* (with respect to w_0) if the following two axioms are satisfied:

> **Axiom 1.** (Monotonicity) For any F, G with $F \subseteq G \subseteq H$ and $w(F) \geq w_0$, $w(F) \leq w(G)$.
> **Axiom 2.** (Locality) For any $F \subseteq G \subseteq H$ with $w(F) = w(G) \geq w_0$ and any $h \in H$, $w(G \cup \{h\}) > w(G)$ implies that also $w(F \cup \{h\}) > w(F)$.

In order to derive a subexponential bound for the expected running time of the Sharir-Welzl algorithm, one more axiom is needed.

> **Axiom 3.** (Basis regularity) Any basis T with $w(T) \geq w_0$ has cardinality exactly $\dim(H, w)$.

An optimization problem (H, w) satisfying Axioms 1–3 (with some w_0) is called a *basis-regular LP-type problem* (with respect to w_0). Let us call any basis B with $w(B) \geq w_0$ a *regular basis.*

Let us see how linear programming fits into this framework. Without loss of generality, we consider the problem of finding the lexicographically smallest point in the intersection of a set H of n halfspaces in R^d. For simplicity, let us assume that the intersection is nonempty (the problem is admissible) and that the optimum is well-defined (it is a point at a finite distance from the origin).

For a subset $G \subseteq H$, we define the value $w(G)$ of G as the optimal vertex (a point of R^d) over the intersection of the halfspaces of G; if the optimum is undefined, we define $w(G)$ as a formal symbol $-\infty$. Hence the set $\mathcal{W}$ is $\{-\infty\} \cup R^d$, where R^d is considered with the lexicographic ordering. This defines an optimization problem (H, w). Let $w_0 \in R^d$ be any lower bound for the optimum value $w(H)$. It is easy to check that (H, w) is a basis regular LP-type problem (with respect to w_0) of combinatorial dimension d.

We are going to describe the Sharir-Welzl algorithm. The algorithm is a recursive function `LpType`(G, T) with two arguments, a set $G \subseteq H$ of constraints, and a basis $T \subseteq G$ (which is not necessarily a basis for G). It computes a basis B for G and uses the set T as an auxiliary parameter for guiding its computations. In order to find a basis for H, `LpType`(H, T) is called, where T is some basis such that (H, w) is a basis-regular LP-type problem with respect to $w(T)$. All bases encountered during the computation of the algorithm will have value at least $w(T)$, so they are regular.

The algorithm uses the following two computational primitives: First, given a (regular) basis $B \subseteq H$ and a constraint $h \in H$, decide whether h violates B, and second, given B and h as above, compute a basis for $B \cup \{h\}$. Both the computational primitives can easily be done in polynomial time for linear programming (violation test in $O(d)$ time and basis change in $O(d^2)$ time if the computation is organized similarly as in the dual simplex algorithm).

Here is a pseudocode for the Sharir-Welzl algorithm:

```
function LpType($G, T$);
    if $G = T$ then
        return($T$)
    else
        choose a random $h \in G \setminus T$;
        $B$ := LpType($G \setminus \{h\}, T$);
        if $h$ violates $B$ then
            $B$ := basis($B \cup \{h\}$);
            return LpType($G, B$)
        else
            return($B$);
        end if;
    end if;
```

The operation

$$B := \texttt{basis}(B \cup \{h\});$$

in the algorithm will be referred to as a *basis change.* The analysis will count the number of basis changes during the computation (the number of violation test is at most n times larger, $n = |G|$).

The following bound is proved in [MSW92] (journal version):

Theorem 2.1 [MSW92] *Let (H, w) be a basis-regular LP-type problem (with respect to some w_0) with n constraints and combinatorial dimension d, let T be a basis with $w(T) \geq w_0$. Then the expected number of basis changes in the call* `LpType`(H, T) *is bounded by*

$$\exp\left(2\sqrt{d \ln \frac{n}{\sqrt{d}}} + O\left(\sqrt{d} + \ln n\right)\right) .$$

□

Let us remark that for n being larger than roughly d^2, it is more advantageous to use a randomized algorithm of Clarkson [Cla88], and the above algorithm is only used as a subroutine in Clarkson's algorithm for subproblems with $n = O(d^2)$. If one could improve the running time bound for $n = O(d^2)$, an overall improvement follows. In our lower bound examples, we will have $n = 2d$.

3 One-permutation and Move-to-front variants

The following is a pseudocode for a "one-permutation" variant of the Sharir-Welzl algorithm, formulated in the spirit of a similar modification of Seidel's algorithm in [Wel91], and also being essentially a dual version of Megiddo's suggestion [Meg92].

```
function OnePermLp($H, T$);
      enumerate the constraints of $H$
         in random order $h_1, \ldots, h_n$;
      $B := T$;
      loop
          if no $h_j \in H$ violates $B$ then
              return($B$)
          else
              $i := \min\{j \in [n];\ h_j$ violates $B\}$;
              $B :=$ basis($B \cup \{h_i\}$)
          end if;
      end loop;
```

The idea of the move-to-front heuristics of [Wel91] is to move the constraint h_i which caused basis change to the beginning of the current permutation of constraints. To implement this in a function `MTF-Lp`, the only change compared to the above code is that the statement

$$B := \texttt{basis}(B \cup \{h_i\});$$

is followed by the command

$$(h_1, h_2, \ldots, h_i) := (h_i, h_1, \ldots, h_{i-1});$$

which moves h_i to the first position and shifts $h_1, \ldots, h_{i-1}$ one position forward.

4 A linear programming example

In this section we define a particular class of linear programs, and we analyze the running time of the Sharir-Welzl algorithm for these linear programs. The analysis can be viewed as an introduction to an abstract example discussed next. The linear programs will also serve as a test example for the one-permutation and move-to-front variants.

Let us consider the linear programming problem in R^d with a set

$$H = \{h_1^0, h_1^1, h_2^0, h_2^1, \ldots, h_d^0, h_d^1\}$$

of $2d$ constraints, defined as follows:

$$\begin{array}{lll} h_1^0: & x_1 \geq 0\,, & \\ h_1^1: & x_1 \geq 1\,, & \\ h_i^0: & x_i \geq 1 - x_{i-1} & i = 2, 3, \ldots, d \\ h_i^1: & x_i \geq x_{i-1} & i = 2, 3, \ldots, d\,. \end{array} \tag{1}$$

We are looking for the lexicographically smallest admissible vector x.

We will present a sequence of simple claims and observations, which lead to modeling the computation of the Sharir-Welzl algorithm by a "flipping process" on vectors of $\{0,1\}^d$.

The optimum lies at the vertex $(1, 1, \ldots, 1)$, and it is determined by the constraints $h_1^1, \ldots, h_d^1$. The other constraints are redundant and they are included in order to confuse the algorithm.

The problem is basis-regular with respect to any finite value, in particular with respect to $w_0 = (0, 0, \ldots, 0)$. Let us investigate the structure of regular bases. For a subset $G \subseteq H$, the optimum is well-defined (bounded in all coordinates) iff G contains at least one h_i^p for every i. The basis for such a G then contains either h_i^0 or h_i^1 for every i.

Let $\mathcal{B}$ stand for the set of all regular bases. $\mathcal{B}$ can thus be identified with $\{0,1\}^d$. Each $B \in \mathcal{B}$ uniquely determines the solution vector $x \in \{0,1\}^d$, which we will denote by $w(B)$. This function is a bijection between $\mathcal{B}$ and $\{0,1\}^d$. For the analysis, it will be more convenient to look at the solution vectors rather than at the bases.

Let $B \in \mathcal{B}$ be a basis, $x = w(B) \in \{0,1\}^d$ the solution vector and $h_i^p \notin B$ a constraint. Then h_i^p violates B iff $x_i = 0$, as one easily checks. If it violates B, then the basis for $B \cup \{h_i^p\}$ is $(B \setminus \{h_i^{1-p}\}) \cup \{h_i^p\}$. Let us see how this basis change changes x. The components $x_1, \ldots, x_{i-1}$ remain unchanged, and x_i changes from 0 to 1. This causes that the r.h.s. of the next constraint in B (the one determining x_{i+1}) changes (either from 0 to 1 or from 1 to 0), and so also x_{i+1} changes its value. Similarly we find that each of $x_{i+2}, \ldots, x_d$ change their value.

These observations imply that the Sharir-Welzl algorithm for our specific problem is equivalent to a "flipping process", which can be described in the form of a recursive procedure `LpFlipping`. This procedure uses one global vector x (the current solution), and it is called with an argument $I \subseteq [d]$, which means the set of "free" indices (corresponding to the indices i with both $h_i^0, h_i^1 \in G$). Initially x is set to some value (corresponding to the solution vector for the initial basis in the Sharir-Welzl algorithm), and the procedure is called with $I = [d]$. The procedure works as follows:

```
procedure LpFlipping($I$);
    if $I = \emptyset$ then
        return
    else
        choose a random $i \in I$;
        LpFlipping($I \setminus \{i\}$),
        if $x_i = 0$ then
            flip($[i..d]$);
            LpFlipping($I$);
        end if;
    end if;
```

The statement flip(J) (with $J \subseteq [d]$) means that for each $j \in J$, the value of x_j is "flipped", that is, $x_j := 1 - x_j$ is executed. The number of basis changes in the Sharir-Welzl algorithm is equal to the number of executions of the "flip" statement in the above procedure, "flips" for short.

Let $T(d, x, I)$ denote the expected number of flips in the call LpFlipping(I), when x is the current vector at the beginning of that call, and let $T(d, x) = T(d, x, [d])$.

We observe that within a recursive call LpFlipping(I), the values of x at positions outside I are insignificant for the number of flips within that call. Hence

$$T(d, x, I) = T(|I|, x|_I)\,, \tag{2}$$

where $x|_I$ denotes the vector of components indexed by I. If $x_1 = 1$, then this can never be changed by subsequent flips, and we find that

$$T(d, x) = T(d-1, x|_{[2..d]}) \;\text{ for } x_1 = 1\,. \tag{3}$$

Another simple observation is that at the moment the call LpFlipping(I) is completed, all the components of $x|_I$ must be 1's.

Let us first investigate $T_0(d) = T(d, (0, \ldots, 0))$. In the call LpFlipping($[d]$), we first get a contribution of $T_0(d-1)$ from the first recursive call, with $I = [d] \setminus \{i\}$. Now the current x has 1 at all positions but possibly x_i. If $i = 1$, then $x_1 = 0$, since x_1 was 0 initially and it was never flipped. In this case, flip($[1..d]$) is executed after the first recursive call, which changes x to the vector $(1, 0, 0, \ldots, 0)$. The subsequent recursive call needs $T_0(d-1)$ expected time by (3).

On the other hand, if $i > 1$, then x_i was always flipped together with x_{i-1}. Since the value of x_{i-1} has changed from 0 to 1 as a result, also x_i must be 1, so there will be no second recursive call for $i \neq 1$. From this we get the recursion

$$T_0(d) = T_0(d-1) + \frac{1}{d}(1 + T_0(d-1))\,.$$

Together with the initial condition $T_0(1) = 1$, this gives $T_0(d) = d$.

Let $T_a(d)$ denote the expected value of $T(d, x)$ with a random choice of x, and $T_w(d)$ the maximum of $T(d, x)$ over all choices of x. The analysis for these quantities is similar to the one for $T_0(d)$. The difference is that x_i can end up as 0 after the first recursive call even if $i \neq 1$. In such case, we have the second recursive call with the initial vector consisting of i ones on the beginning followed by $d - i$ zeros, which by (3) needs $T_0(d-i)$

expected time. In the worst-case, having no information whether x_i will be 0 or 1 after the first recursion is completed, we assume that $x_i = 1$. This gives $T_w(1) = 1$,

$$T_w(d) \le T_w(d-1) + 1 + \frac{1}{d}\sum_{i=1}^{d-1} T_0(d-i) = T_w(d-1) + \frac{d+1}{2}$$

or $T_w(d) \le \frac{d(d+3)}{4}$.

For a randomly chosen x, x_i is independent of x_{i-1} on the beginning, and since it is flipped simultaneously with x_{i-1} during the first recursive call and x_{i-1} ends up as 1, the value of x_i after the first recursive call will be 0 or 1 with equal probability. This gives $T_a(1) = 1/2$,

$$T_a(d) = T_a(d-1) + \frac{1}{2} + \frac{1}{2d}\sum_{i=1}^{d-1} T_0(d-i) = T_a(d-1) + \frac{d+3}{4},$$

that is, $T_a(d) = \frac{d(d+3)}{8}$.

5 A slow abstract example

By generalizing the linear programming example from the previous section, we obtain an example where the Sharir-Welzl algorithm can be proved to run in expected time close to the known upper bound. We cannot model the example by a linear programming problem; rather it will be a LP-type problem (resembling boolean programming) fitting into the Sharir-Welzl abstract framework.

Let A be a lower triangular $d \times d$ matrix with zero diagonal over Z_2 (the two-element field, $\{0, 1\}$ with addition and multiplication modulo 2). An LP-type problem P_A defined by A will be as follows.

For $i = 1, 2, \ldots, d$ and $p = 0, 1$, let $h_i^p = h_i^p(A)$ denote the constraint

$$x_i \ge \sum_{j=1}^{i-1} a_{ij} x_j + p\,.$$

Here $x_1, \ldots, x_d$ denote variables over Z_2. The addition and multiplication on the r.h.s. is in Z_2, and the inequality is to be interpreted as follows: If the r.h.s. evaluates to 1 for given values of x_j, then the constraint requires that x_i is also 1, otherwise it puts no requirement on x_i.

The constraint set of the problem P_A will be $H_A = \{h_i^p(A);\ i \in [d], p \in Z_2\}$.

The range of the value function w will be the set $W = \{-\infty\} \cup Z_2^d$. The ordering on W is defined as follows: $-\infty$ is the smallest element, and Z_2^d is ordered lexicographically (taking $0 < 1$ in Z_2).

For a subset $G \subseteq H_A$, the value $w_A(G)$ is defined as $-\infty$ if there is an index $i \in [d]$ such that G contains neither h_i^0 nor h_i^1, otherwise $w_A(G)$ is the lexicographically smallest vector $x \in Z_2^d$ satisfying all the constraints of G.

We leave it to the reader to check that this defines a basis-regular LP-type problem (with respect to $w_0 = (0, 0, \ldots, 0)$) of combinatorial dimension d. A regular basis is a subset of H containing exactly one constraint of h_i^0, h_i^1 for each i. Let $\mathcal{B}$ be the set of all regular bases.

The problem discussed in the previous section can be viewed as a special case of P_A, with $a_{i,i-1} = 1$ and $a_{ij} = 0$ otherwise. The following facts mentioned in connection with this special case hold in general:

The optimal value of P_A is $(1, \ldots, 1)$. The function w_A restricted to $\mathcal{B}$ defines a bijection between $\mathcal{B}$ and the set Z_2^d of solution vectors. Let $B \in \mathcal{B}$ be a basis and $h_i^p \notin B$ a constraint. A constraint $h_i^p \notin B$ violates B iff $x_i = 0$, with $x = w_A(B)$, and then the basis for $B \cup h_i^p$ is $(B \setminus \{h_i^{1-p}\}) \cup \{h_i^p\}$.

Describing the change in the solution vector caused by such a basis change is somewhat more complicated this time. First of all, x_i becomes 1. Then the r.h.s. of $h_{i+1}^{p_{i+1}}$, the constraint of B defining x_{i+1}, changes its value iff $a_{i+1,i} \neq 0$, and in such case x_{i+1} is also flipped, otherwise it remains unchanged. For x_{i+2}, we again look at the r.h.s. of the constraint in B defining it. Its value could change because of x_i, and also because of x_{i+1}, or these two influences can cancel out. However, this condition (for changing x_{i+2}) only depends on $a_{i+1,i}, a_{i+2,i}$ and $a_{i+2,i+1}$, *not* on the values of x_{i+1}, x_{i+2}.

Proceeding with this analysis inductively, we define quantities f_{ik} for $k > i$, with the meaning that f_{ik} is 1 iff x_k will be flipped when a basis change occurs at index i. We get that f_{ik} is determined as follows:

$$f_{ik} = a_{ki} + a_{k,i+1} f_{i,i+1} + a_{k,i+2} f_{i,i+2} + \cdots + a_{k,k-1} f_{i,k-1} \tag{4}$$

(the addition is modulo 2). Let $F = F(A)$ denote the upper triangular matrix of the f_{ik}'s; we will call it the *flipping matrix*.

We thus find that the computation of the Sharir-Welzl algorithm with an initial basis $B \in \mathcal{B}$ can be modeled by the following procedure `A-Flipping` called with the initial value of x set to $w_A(B)$ (the procedure is identical to `LpFlipping` except for the `flip` statement):

```
procedure A-Flipping($I$);
      if $I = \emptyset$ then
          return
      else
          choose a random $i \in I$;
          A-Flipping($I \setminus \{i\}$),
          if $x_i = 0$ then
              flip($\{i\} \cup \{k \in [i+1..d];\ f_{ik} = 1\}$);
              A-Flipping($I$);
          end if;
      end if;
```

Our goal is to analyze the expected number of flips, the expectation being over the internal random choices of the algorithm, a random choice of the initial vector x and a random choice of the matrix A (the relevant entries being independent uniformly distributed 0/1 random variables).

We see that the matrix A enters the procedure only via the F matrix. We have the following elementary lemma:

Lemma 5.1 *Let a_{ij} $(2 \leq i \leq d,\ 1 \leq j < i)$ be independent uniformly distributed 0/1* *variables. Then f_{ik} $(1 \le i \le d-1,\ i < k \le d)$ defined by (4) are also independent uniformly distributed 0/1 variables.*

Proof: The value of f_{ik} is determined by the values of $a_{\ell j}$ with $i \le j < \ell \le k$ (this can either be derived formally from (4) or seen from the considerations leading to that formula). Let $i', k', i' < k'$ be indices such that $k' - i' < k - i$, or $k' - i' = k - i$ and $k' < k$. Then we have that the value of $f_{i'k'}$ does not depend on the value of a_{ik}. Then also f_{ik} determined by the formula (4) is independent of any combination of the $f_{i'k'}$'s (with i', k' as above), because a_{ik} is independent of these (a 0/1 quantity independent of a_{ik} is added to a_{ik}, so the result remains independent of anything a_{ik} was independent of). $\square$

We can thus replace the expectation over the choice of A by expectation over the choice of F. Let $T(d, F, x, I)$ be the expected number of flips in the call `A-Flipping`(I) with flipping matrix F and initial vector x. We let $T(d, F, x) = T(d, F, x, [d])$; again we observe that $T(d, F, x, I) = T(|I|, F|_I, x|_I)$, $F|_I$ meaning the submatrix of elements of F with both row and column indices in I . Further we let $T(d)$ be the expectation of $T(d, F, x)$ over independent and random choice of x and F.

With random x and F, the first recursive call in procedure `A-Flipping` requires $T(d-1)$ expected time. Let us consider the situation after the first recursive call is finished. If i is the randomly chosen index, we have $x_j = 1$ for $j \ne i$. We claim that the current value of x_i is an uniformly distributed 0/1 random variable, which is independent of i and of the random choices of the algorithm during the first recursive call, and also of F. If $i = 1$, the value of x_1 has not changed in the first recursive call, so the claim holds by independence of x_1 and F. For $i > 1$, x_i has been flipped some number of times during the first recursive call. The number of these flips did not depend on the initial value of x_i, hence also the resulting value is random and independent of F, so the claim holds.

If $x_i = 1$, the procedure is thus finished. If $x_i = 0$, the `flip` statement is executed, which yields the current vector x with $x_j{=}1$ for $1 \le j \le i$ and $x_j = 1 - f_{ij}$ for $i < j \le d$. Then we have the second recursive call, which requires expected time (using a generalization of (3)) $T(d-i, F|_{[i+1..d]}, x|_{[i+1..d]})$. The entries of F are independent and $x|_{[i..d+1]}$ is defined by ith row of F. Hence the initial vector $x|_{[i+1..d]}$ and the flipping matrix $F|_{[i+1..d]}$ are random and independent, and so the expected time for the second recursive call is equal to $T(d-i)$.

Together with the initial condition $T(1) = 1/2$ we get the recursion

$$T(d) = T(d-1) + \frac{1}{2} + \frac{1}{2d}\sum_{i=1}^{d-1} T(d-i)\,.$$

Our considerations thus show that for every d there is a specific matrix F such that for a random choice of the initial basis, the expected number of flips is at least $T(d)$. Translated back to the LP-type problems setting, it means that for some A the problem P_A requires at least $T(d)$ basis changes in the Sharir-Welzl algorithm (started with a randomly chosen initial regular basis).

It remains to give an asymptotic estimate for $T(d)$, which is more or less routine. We first substitute $b_d = T(d) + 1$, obtaining the relations

$$b_0 = 1\,, \qquad b_d = b_{d-1} + \frac{1}{2d}\sum_{i=0}^{d-1} b_i\,.$$

From this we derive an equation for the generating function $b(z) = b_0 + b_1 z + b_2 z^2 + \ldots$,

$$b(z) = 1 + zb(z) + \int_0^z \frac{b(t)}{2(1-t)} \mathrm{d}t$$

whose solution is

$$b(z) = \frac{\exp\left(\frac{z}{2(1-z)}\right)}{1-z}.$$

Using the Maple V software, with the asymptotic analyzer developed by B. Salvy, P. Flajolet and others (see [FSZ91]), the following asymptotics for b_d is obtained:

$$b_d = \frac{e^{\sqrt{2d}}}{e^{1/4} 2^{3/4} \sqrt{\pi} \sqrt[4]{d}} + O\left(\frac{e^{\sqrt{2d}}}{d^{3/4}}\right).$$

This is in a good agreement with numerically computed values of b_d. If one does not believe in the automatic asymptotic analyzer, one can at least prove a less precise estimate by induction.

We can summarize our considerations in a theorem:

Theorem 5.2 *For every d there exists a basis-regular LP-type problem of combinatorial dimension d with $2d$ constraints (of the form P_A), such that for a randomly chosen initial d-element basis the expected running time of the Sharir-Welzl algorithm is $\Omega(e^{\sqrt{2d}}/\sqrt[4]{d})$.* $\square$

6 One-permutation and Move-to-front: Experimental results

We saw that our linear programming example (1) gives no impressive lower bounds for the Sharir-Welzl algorithm. However, it seems to have another interesting property: it makes the one-permutation variant (discussed in Section 3) quite slow, and the move-to-front variant even slower. At present, we do not know how to analyze the expected number of basis changes theoretically for either of these variants. We have neither a superpolynomial lower bound nor a subexponential upper bound resembling the one known for the Sharir-Welzl algorithm in general.

Methods. We have performed computer simulations of the one-permutation and move-to-front variants. In both cases, the computation of the algorithm was first reformulated as a flipping process on 0/1 vectors (somewhat more complicated than the one for the Sharir-Welzl algorithm).

For the one-permutation variant, Emo Welzl suggested a binary tree data structure for both quick detection of the position for flipping and executing the flipping quickly. The data structure only needs $O(\log d)$ time per basis change in simulation of `OnePermLp` on example (1) (the current vector is only represented implicitly in the data structure). The algorithm was then coded independently by David Alberts and by the author, and the correctness of the implementation was moreover verified by comparison with a slower but more straightforward implementation.

For the move-to-front variant, it was not obvious how to apply a similar idea to speed up the simulation, so a straightforward implementation with $O(d)$ time per basis change was used. The code was written by the author. All implementations were made in the C++ language, as nothing better was readily available.

The random permutations needed by the algorithms were generated using the `drand48` random number generator of the Unix system. The following well-known method was used: initialize a sequence to $1, 2, \ldots, d$, and then for $i = 1, 2, \ldots, d$, exchange the ith element with jth, j a random index in range $[i..d]$. Some of the tests were in fact performed with doubly shuffled permutations (the above algorithm was once more applied on the first random permutation as an initial sequence), but no significant differences appeared in comparison with permutations shuffled only once.

The experiments were conducted on SPARC workstations. The results for the largest dimensions shown usually needed several days of computing time.

Various numbers of repetitions for every dimension were used: 10^3, 10^4 and 10^5. For the values where 10^4 repetitions were still computationally feasible, the results show no substantial difference between 10^4 and 10^3 repetitions, thus one can believe that the results are close to the true expectation. For larger dimensions, where only 10^3 repetitions were performed, the results may be less reliable. This issue will be discussed later.

For the one-permutation variant, the basis $\{h_1^0, h_2^0, \ldots, h_d^0\}$ was set as the initial one, corresponding to the vector $(0, 1, 0, 1, \ldots)$. This is the "opposite" of the basis defining the optimum, so one would expect it should cause the algorithm to run slowly. Some experiments were also made with a random initial basis. The resulting mean of running times was somewhat smaller, but the general trend of growth was very similar.

For the move-to-front variant, a randomly generated initial basis was used.

Results. The results for the one-permutation variant are depicted in Figure 1. The vertical axis, corresponding to the mean of the number of basis changes over the indicated number of repetitions, is in logarithmic scale. This figure shows that the observed values lie close to the curve $e^{\sqrt{d}}$, but seem to grow somewhat slower. Thus `OnePermLp` is drastically slower than the Sharir-Welzl algorithm itself (the expected running time $T_a(d)$ of the Sharir-Welzl algorithm for a random initial basis is shown in the picture for comparison), but still seems to fit below the theoretical upper bound valid for the Sharir-Welzl algorithm.

Figure 2 depicts the results for the move-to-front strategy in a similar way. The growth in this case is apparently much faster, perhaps exponential.

7 Discussion and open problems

The main problem is of course to find a better combinatorial linear programming algorithm (ideally a strongly polynomial one), or prove a lower bound. It seems that there is still hope to show that the Sharir-Welzl algorithm performs better on actual linear programs, using some property of linear programming not reflected in the abstract framework. It would be interesting to analyze the one-permutation variant, at least for our particular example. Quite recently Gärtner [Gär92] found a subexponential randomized algorithm for a much wider class of optimization problems than the ones fitting into the Sharir-Welzl framework. It is likely that our example could be used to demonstrate that Gärtner's bound for his algorithm is essentially the best possible under his axioms. But it has not been done yet and it looks somewhat more complicated than for the Sharir-Welzl algorithm.

Other problems are inspired by the computer simulations performed.

Figure 3 shows the distribution of the number of basis changes (roughly corresponding

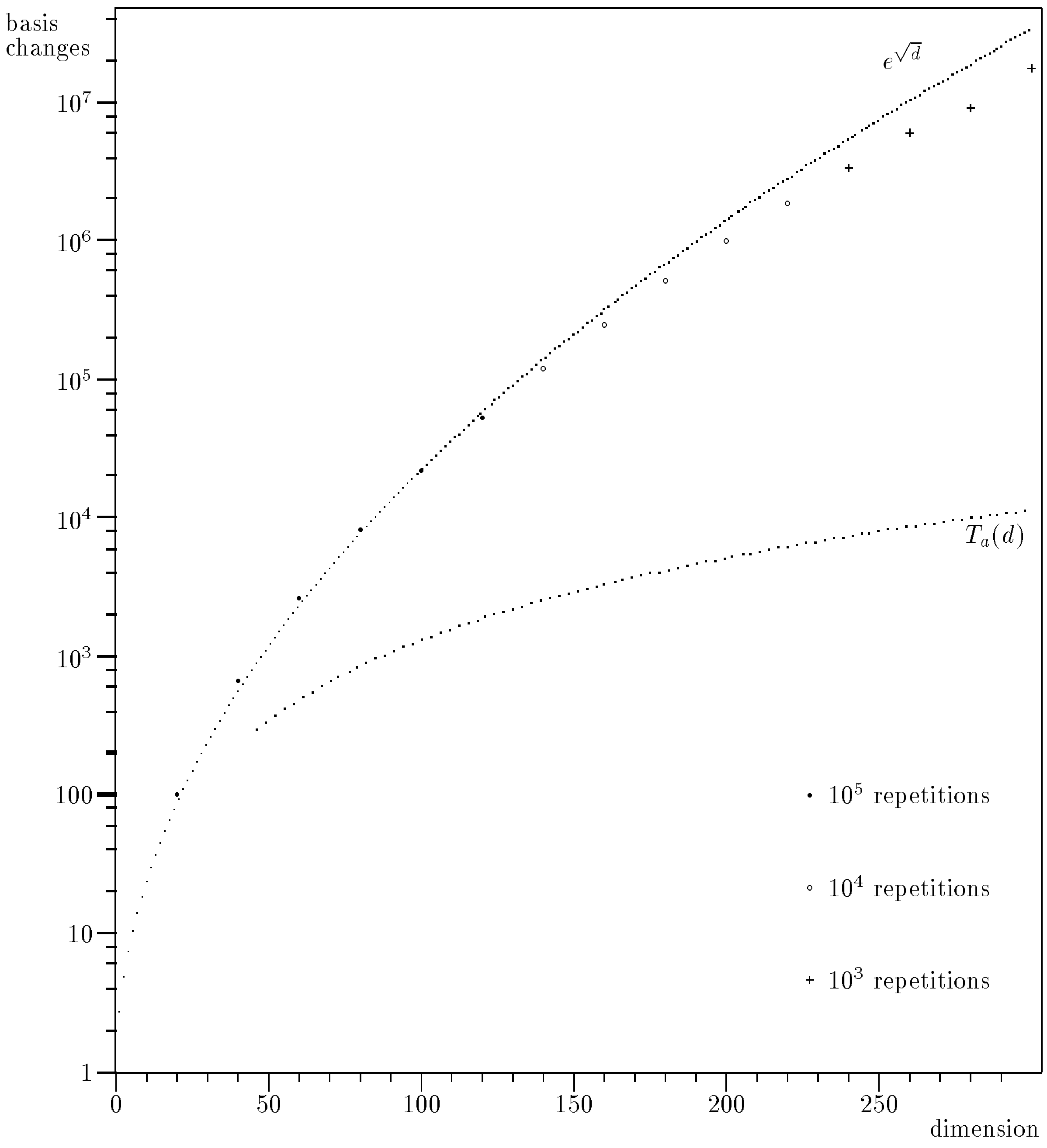

Figure 1: Simulation of the one-permutation variant on example (1); initial vector $(0,1,0,1,\ldots)$.

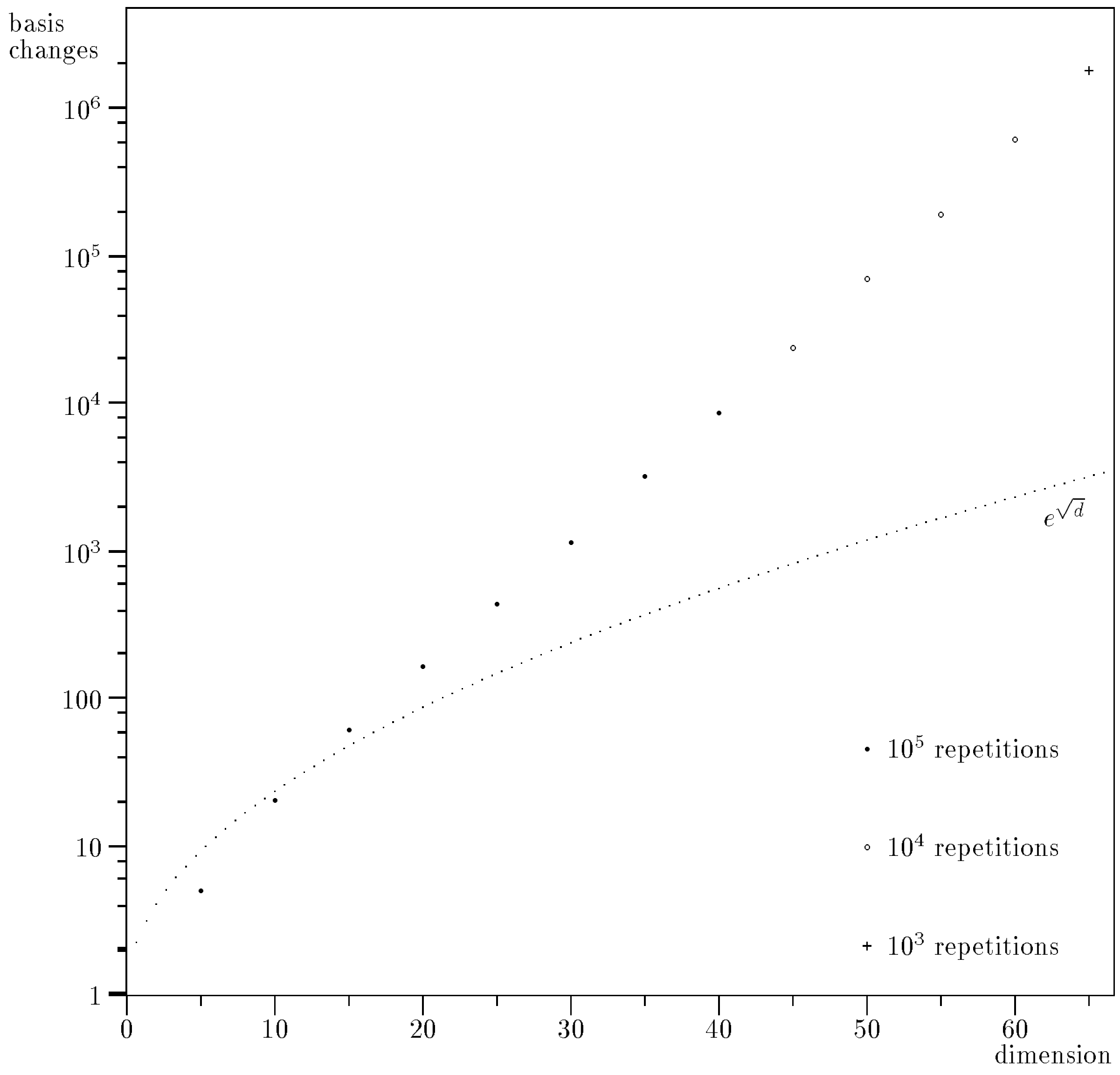

Figure 2: Simulation of the move-to-front variant on example (1); random initial basis.

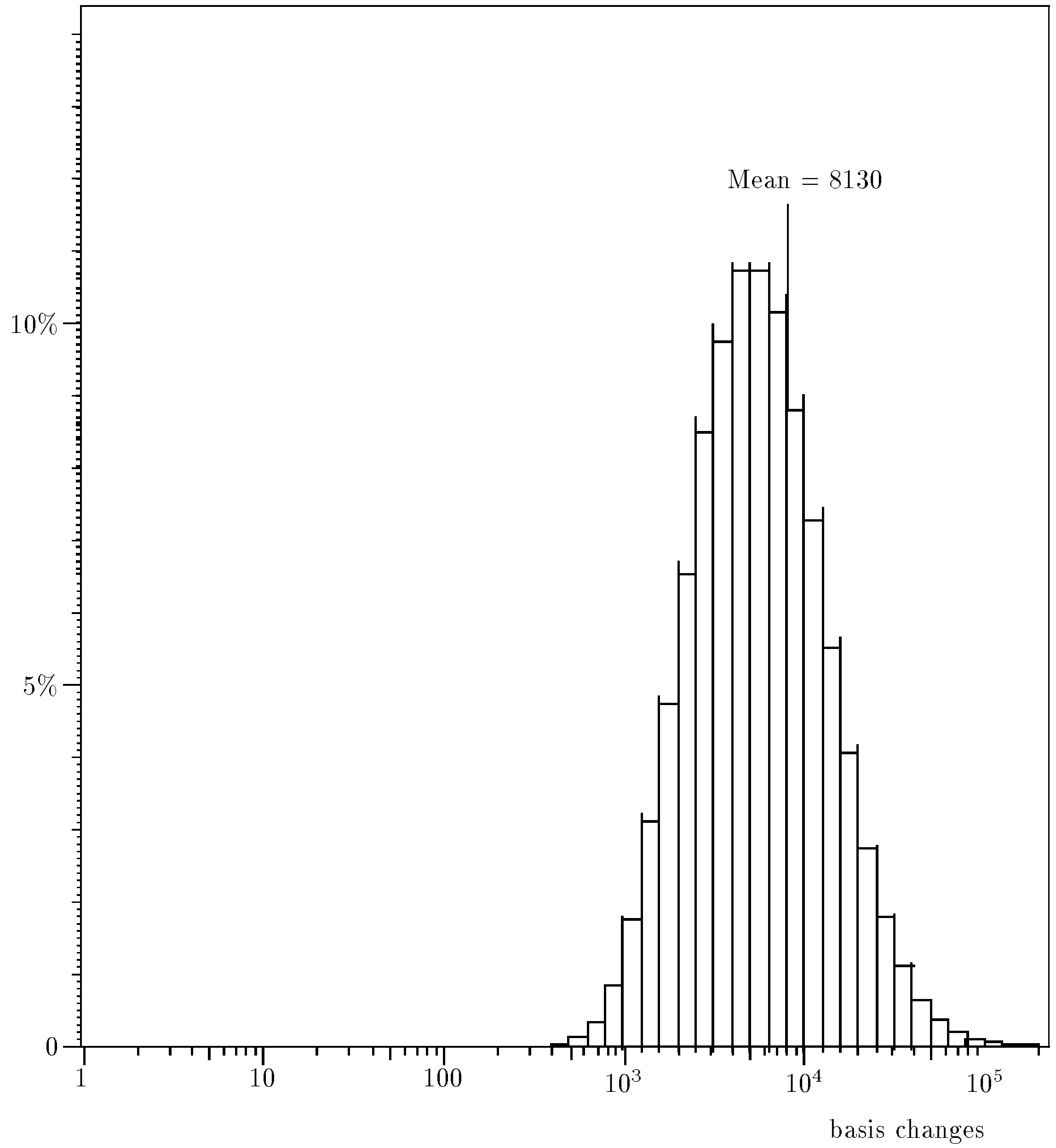

Figure 3: Simulation of the one-permutation variant on example (1), $d = 80$: distribution of the number of basis changes (based on 100,000 repetitions).

to running times) for the one-permutation variant on example (1), for dimension 80. This figure reveals an interesting aspect: Most of the runs give results far below the mean, and only small proportion of long runs is responsible for a large part of the mean. Specifically, the mean number of basis changes is over 8000, while the most probable number of basis changes (maximum of the distribution curve) is about 5000. About 68% of the runs are below the mean. It turns out that for this algorithm, one can get a somewhat better expected running time by a "restarting strategy": if the running time of the current run has exceeded certain threshold, terminate it and restart.

This effect may seem somewhat peculiar on the first sight. One can demonstrate it on a simple artificial example. Suppose that certain algorithm A runs for 1 time unit with probability $1-p$, and for K time units with probability p. Then the expected running time is $Kp+(1-p)$. If we run A for one time unit and restart it if it does not stop, we get an algorithm with expected running time $(1-p).1+(1-p)p.2+(1-p)p^2.3+\cdots=1/(1-p)$; for p small and Kp large this is an drastical improvement. It would be interesting to investigate which algorithms can be made faster by this kind of restarting (perhaps also embedded in a recursive procedure).

Another aspect of the distribution of the running times is that a very large number of repetitions is needed in order to estimate the expectation with a reasonable reliability. In the example of the previous paragraph, if we make fewer runs than $1/p$, we probably never see a long run and we might falsely conclude that the expected running time of algorithm A is 1. Similarly for the one-permutation variant, the distribution is such that we seldom see a run much longer than the mean, although such runs influence the mean significantly. Thus experiments with a small number of repetitions are inadequate. We believe that in our experiments, the number of repetitions was already sufficient to get means close to the true expectation (at least for dimensions where experiments with 10^3 and 10^4 or even 10^5 repetitions were conducted, the outcomes differed only by a little), but there is no absolute guarantee for this.

This brings us to another issue — pseudorandom numbers. Little is known about the influence of replacing random numbers by pseudorandom ones in various randomized algorithms. We have performed some experiments with "artificially bad" random number generators (see [Knu73] for a thorough discussion of bad generators). In the algorithms we considered, a large part of the expected running time comes from a small portion of "exceptional" runs, and this could perhaps make them sensitive to the random number quality.

Test runs of the one-permutation variant simulations were made with the "Fibonacci" generator (the next pseudorandom number is the sum of the previous two ones modulo a fixed number, 2^{15} in our case), and with a linear congruential generator of poor quality (with a small multiplier, equal to 91, and modulus 2^{15}). These preliminary studies did not show any drastical deviation of the results from the (presumably) good random number generator, but there seemed to be some systematic deviations. The results for the Fibonacci generator are given in Table 1. The table shows the presumably "true" expectation (the mean of 10^5 runs for the Unix random number generator), and the deviations for 10^3 and 10^4 runs with the Unix generator and with the Fibonacci generator (in %).

For the 10^3 runs of the Unix generator, the results are systematically somewhat smaller than for 10^5 runs, which can be explained by the distribution of the running times, see above. The means gained by the Fibonacci generator seem to behave rather differently; for some dimensions (60,120) they seem to "converge" to some value larger than the true

Dimension	Mean of 10^5 runs	Deviation: Unix generator 10^3 runs	Deviation: Unix generator 10^4 runs	Deviation: Fibonacci generator 10^3 runs	Deviation: Fibonacci generator 10^4 runs
20	102.2	−1.2%	+0.4%	+0.9%	+2.7%
40	667.8	−0.9%	+0.6%	+1.6%	+1.0%
60	2635	−3.7%	−0.6%	+2.7%	+4.4%
80	8166	−3.3%	+1.6%	+3.8%	+3.4%
100	21760	−5.7%	−1.0%	−2.1%	+0.5%
120	52600	−0.5%	−0.7%	+11.4%	+10.4%

Table 1: A comparison with the results obtained using a "bad" random number generator (one-permutation variant).

expectation. The results for the other bad generator (not shown) indicate a similar tendency, sometimes deviating downwards. Could one extrapolate this and claim something like that with a given "quality" of the generator (presumably bounded from above by the word length used) there is no hope in empirically estimating the expectation with accuracy better than some lower bound, no matter how many repetitions we perform? So far this is a pure hypothesis, and a much more thorough study would be needed to evaluate the significance of such deviations. Our preliminary experiments only indicate that this issue might be worth considering; presumably one should start with some algorithm or random process where good theoretical results on the distribution are available.

Acknowledgement. Emo Welzl invented a nice and efficient data structure for modeling the one-permutation variant. I thank him for this and also for several useful discussions. Jan Sochor worked on analyzing the flipping processes `LpFlipping` and its relatives. He independently obtained partial results concerning the analysis of T_0 and T_a, and I used some of his ideas in the presentation. Last but not least, I thank David Alberts for implementing some of the algorithms, performing experiments and preprocessing the results.

References

[Cla88] K. Clarkson. Las Vegas algorithm for linear programming when the dimension is small. In *Proc. 29. IEEE Symposium on Foundations of Computer Science*, pages 452–457, 1988 (preliminary version; a later version with improved results is a manuscript from 1989).

[FSZ91] P. Flajolet, B. Salvy, and P. Zimmerman. Automatic average-case analysis of algorithms. *Journal of Symbolic Computation*, 1991.

[Gär92] B. Gärtner. A subexponential algorithm for abstract optimization problems. In *Proc. 33. IEEE Symposium on Foundations of Computer Science*, 1992. To appear.

[Kal92] G. Kalai. A subexponential randomized simplex algorithm. In *Proc. 24. ACM Symposium on Theory of Computing*, 1992.

[Kha80] L. G. Khachiyan. Polynomial algorithm in linear programming. *U.S.S.R. Comput. Math. and Math. Phys.*, 20:53–72, 1980.

[Knu73] D. E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms.* Addison-Wesley, 1973.

[Meg92] N. Megiddo. A note on subexponential simplex algorithms. Lecture at 2. Israeli Computational Geometry Workshop, Eilat, April 1992.

[MSW92] J. Matoušek, M. Sharir, and E. Welzl. A subexponential bound for linear programming. In *Proc. 8. ACM Symposium on Computational Geometry*, pages 1–8, 1992.

[Sei91] R. Seidel. Small dimensional linear programming and convex hulls made easy. *Discrete & Computational Geometry*, 6(5):423–434, 1991.

[SW92] M. Sharir and E. Welzl. A combinatorial bound for linear programming and related problems. In *Proc. 1992 Symposium on Theoretical Aspects of Computer Science (Lecture Notes in Computer Science 577)*, pages 569–579. Springer-Verlag, 1992.

[Wel91] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science (Lecture Notes in Computer Science 555)*, pages 359–370. Springer-Verlag, 1991.