Runtime Prediction of Real Programs on Real Machines*

Ulrich Finkler[†] Kurt Mehlhorn[‡]

Abstract

Algorithms are more and more made available as part of libraries or tool kits. For a user of such a library statements of asymptotic running times are almost meaningless as he has no way to estimate the constants involved. To choose the right algorithm for the targeted problem size and the available hardware, knowledge about these constants is important.

Methods to determine the constants based on regression analysis or operation counting are not practicable in the general case due to inaccuracy and costs respectively. We present a new general method to determine the implementation and hardware specific running time constants for combinatorial algorithms. This method requires no changes of the implementation of the investigated algorithm and is applicable to a wide range of of programming languages. Only some additional code is necessary.

The determined constants are correct within a constant factor which depends only on the hardware platform. As an example the constants of an implementation of a hierarchy of algorithms and data structures are determined. The hierarchy consists of an algorithm for the maximum weighted bipartite matching problem (MWBM), Dijkstra's algorithm, a Fibonacci heap and a graph representation based on adjacency lists. The errors in the running time prediction of these algorithms using exact execution frequencies are at most 50 % on the tested hardware platforms.

1 Introduction

Big-O analysis of algorithms is concerned with the asymptotic analysis of algorithms, i.e., with the behavior of algorithms for large inputs. It does not allow the prediction of actual running times of real programs on real machines and therefore its predictive value is limited.

• An algorithm with linear running time $O(n)$ is faster than an algorithm with running time $O(n^2)$ for sufficiently large n. Is $n = 10^6$ large enough? Asymptotic analysis of algorithms is of little help to answer this question. It is however true that a well-trained algorithms person who knows program and analysis can make a fairly good guess.

• Algorithms are more and more made available as part of software libraries or algorithms tool kits, LEDA is a widely used example [10]. For a user of such a library statements of asymptotic running times are almost meaningless as he has no way to estimate the constants involved. After all, the purpose of a tool kit is to hide the implementations from the end user.

The two items above clearly indicate that we need more than asymptotic analysis in order to have a theory with predictive value. *The ultimate goal of analysis of algorithms must be a theory that allows to predict the actual running time of an actual program on an actual machine* with reasonable precision (say within a factor of two). We must aim for the following scenario: When a program is installed on a particular machine a certain number of well-chosen tests is executed in order to learn about machine parameters relevant for the execution of the program. This knowledge about the machine is combined with the analysis of the algorithm to predict running time on specific inputs. In the context of an algorithms library one could even hope to replace statements about asymptotic execution times by statements about actual execution times during installation of the library.

Asymptotic analysis of algorithms approximates the running time $T(x)$ of an algorithm on input x as

$$T(x) \approx \sum_i C_i f_i(x) \qquad (1.1)$$

where $f_i(x)$ measures the frequency of execution of a certain set of operations M_i and C_i measures the execution time of M_i on an idealized Random Access Computer. The theoretical analysis provides approximations for the execution frequencies $f_i(x)$, for example in the worst or average case.

Due to computer architectures with registers, caches, pipelines etc. the execution time of a subset M_i of the code is context sensitive and hence the execution time is

$$T(x) = \sum_i \sum_{1 \le j \le f_i(x)} C_{i,j,x}$$

*Supported by ALCOM-IT and Graduiertenkolleg 'Effizienz und Komplexität von Algorithmen und Rechenanlagen', Universität Saarbrücken, Germany.

[†]Max-Planck-Institut für Informatik, Saarbrücken.

[‡]Max-Planck-Institut für Informatik, Saarbrücken.

where $C_{i,j,x}$ is the time for the j-th execution of M_i on input x, taking into account whether or not data are present in the cache, for example. The coefficients $C_{i,j,x}$ may vary by a factor F, which is the quotient between the fastest and slowest possible execution of an instruction. The execution time for a single instruction may vary widely, e.g. on one of the architectures used for experiments (Pentium with 8-3-3-3-burst and 60 MHz external clock) F is close to 30 for a 200 MHz CPU (the burst access to main memory takes about 130 ns to transfer the first 64 bit package which is 26 times the clock cycle of the processor). Nevertheless we will argue in section 3 that the execution time of many interesting programs can be approximated as in equation (1.1) within a constant factor $\bar{F}$ which is much smaller than F. Moreover, the set of necessary constants C_i is fairly small, even for complex algorithms [6] and can be determined with automated experiments, executed once for each platform.

There are basically two approaches for the determination of the C_i in the literature:

Regression analysis: It is easy to instrument a program such that the total running time $T(x)$ and the execution frequencies $f_i(x)$ are determined during a program run ([6],[8]). It is therefore tempting to measure $T(x)$ and $f_i(x)$ for a large number of inputs x and to determine the constants C_i by regression analysis. We will argue in section 4.1 that this approach is unsound in general because of systematic measurement errors. For example, we should expect the C_i to systematically depend on input size, e.g., due to cache misses. However, regression analysis is only meaningful when measurement errors are statistically distributed.

Operation counting: This technique basically counts the operations of a program and estimates the time for the execution of the underlying assembler structure. [2] describes a technique called 'mem-counting', which charges memory references by insertions of counter instructions into the source code for each such instruction. Methods based on operation counting provide feasible running time predictions, but they are costly to automate (section 4.2).

In this paper we present a new approach for the automated determination of the coefficients C_i, which we call *timing of equivalent code fragments*. We will argue, that it is possible to design experiments which execute a fragment in isolation (so that the interpretation of the timing needs no regression analysis), which execute the fragment approximately within its real context (so that the timing reflects the actual running time) and which execute the fragment many times (so as to minimize the effect of measurement errors).

In order to reach these three goals the experiments time slightly modified code fragments. The fragments are similar to the original as far as the executed operations and the locality of memory references are concerned. We calls such fragments *equivalent code fragments.*

As a test for our approach we have chosen an implementation in C of a hierarchy of algorithms and data structures. The top level consists of an algorithm for the maximum weighted bipartite matching (MWBM) problem which uses Dijkstra's algorithm as a subroutine. The implementation of Dijkstra's algorithm is based on a Fibonacci heap. Both algorithms use a graph representation based on adjacency lists.

This test set has many properties which complicate running time prediction. It consists of several levels of algorithms, only an amortized analysis of the costs is possible and the execution frequencies depend strongly on the distribution of the edge weights as well as the order in the adjacency lists, not only on the number of edges and nodes. Additionally, the Fibonacci heap is a fairly complicated data structure which uses heavily dynamic memory allocation for small objects.

We performed experiments on a variety of machines with Pentium (CISC) and Sparc (RISC) architecture and different operating systems (Linux, SunOS and Solaris) (section 4.3.1,page 6). We used random inputs for the experiments, the execution frequencies of each instance were determined experimentally. The input sizes varied from 1000 up to 100000 nodes and 1000 to 400000 edges. The actual execution times varied from a few milliseconds up to several minutes. In all cases the measured time was within 50 % of the predicted time (and usually closer, fig. 3, page 9). The actual execution times varied from a few milliseconds up to several minutes. Moreover, for the determination of the running time constants inputs with less than 50000 nodes or edges were sufficient, since the constants reach asymptotically a maximum once memory usage exceeds the size of the cache. Thus our method allows not only *interpolation*, but also *extrapolation.*

Due to the arguments in section 3 and the experimental results it is likely that the method will produce useful results for combinatorial algorithms in general. The identification of code fragments and their replacement by equivalent code fragments is a potential source of error. We implemented redundant experiments for several code segments and found that different experiments produced similar running time coefficients. We conclude that the method is robust.

Additionally, the hierarchical structure demonstrates the compatibility of our approach with the hierarchical structure of algorithms. The constants of subalgorithms can be reused in the running time prediction as the methods in implementations.

Running time constants can be combined with any kind of theoretical analysis (known execution frequencies, best case frequencies, worst case frequencies, average case frequencies) to make predictions. If the theoretical analysis provides the execution frequencies within a constant factor, time prediction is possible within a constant factor. In the other cases the error in the time prediction is dominated by the error of the theoretical analysis. An example is given with an average case analysis of Dijkstra's algorithm (page 8). Additionally the constants evaluated with our method allow comparisons between different implementations and hardware.

2 Code Fragments

How does one find feasible code fragments to be used in equation (1.1)? A code fragment is a piece of straight-line code with 'holes'. The identifying property of code fragments is that execution of the program executes every instruction of the fragment the same number of times. Due to this fact feasible code fragments are defined quite naturally as loop bodies, branches of conditionals and function bodies. The holes correspond to nested fragments, e.g. nested loop or nested function bodies.

Frequently one may merge code fragments into larger units without violating this property, e.g. when both branches of a conditional execute an approximately equal set of instructions or a function call has constant execution time. We found identification of feasible code fragments a fairly straightforward task.

3 Context Sensitivity

As mentioned, the execution time of an instruction depends on its context. In this section we will show, that for many interesting programs the variation of the execution time of a code segment due to caching and pipelining can be expected to be much smaller than the worst case variation of the execution time of a single instruction.

We discuss caching first. We call a reference to memory *local* if no cache miss occurs, and *nonlocal* otherwise. The 90/10 rule [3] states that a program executes 90 % of its instructions in 10 % of its code, which we call the 'core' of the program. An example calculation in [3] assumes, that only the fraction of the core which fits into the cache simultaneously produces no cache misses. But the experimental results in the same book show much better cache hit rates. Already 1 kB 2-way associative cache reduces the cache misses on a Unix machine to 20 % of the instructions.

This result is not surprising. In most programs the code of inner loops is smaller than a few thousand bytes. The core of a program consists of many small pieces which are executed in the cache one after the other. Therefore it is realistic to assume that 80-90 % of the instructions are local. So a highly associative cache (2-way or more), that is bigger than the average loop size, gives a minimum hit rate of about 80 % for instructions. By the above, we should expect this hit rate independent of program size.

Data access is less local than instruction access [3]. However, even for data references there are frequently at least two local data references for every nonlocal reference. For example, for a nonlocal access to an array element, the access to the base address of the array and to the offset are usually local. The experimental data in [3] confirm, that at least 50 % of the data references are local.

In the case of some kB highly associative cache and ordinary programs the following rules of thumb are plausible:

1. There are at least as many code references as data references.
2. At least 50 % of the data references are local.
3. At least 80 % of the code references are local.

Consider a sequence of n_c accesses to instructions and n_d accesses to data, and let $n = n_c + n_d$. Under the assumptions above the maximum experienced slowdown δ due to cache misses is

$$\begin{aligned} \delta &= \frac{(\frac{4}{5}+\frac{1}{5}F)n_c + (\frac{1}{2}+\frac{1}{2}F)n_d}{n} \\ &\leq \frac{7}{20}F + \frac{13}{20} \end{aligned} \tag{3.2}$$

For $F = 30$, we have $\delta \leq 11.15$. We will next argue that the actual factor is much smaller for equivalent code fragments.

A code fragment, that belongs to a leading term in the running time, is executed many times. It will be embedded in a loop or recursion. After a certain number of repetitions, the context of the loop is dominated by the loop itself. Together with an input for the fragment which produces a similar locality of code and data references, caching efficiency between experiment and original will not differ too much. For a similar context the efficiency of a pipeline is similar too.

There are cases, in which the context cannot be reproduced well. These are code fragments, which are executed multiple times, but between their executions intermediate code is executed. But already with an instruction cache of a few kB either the probability, that the fragment is still present in the cache at the time of the next execution, is high or the intermediate code dominates the running time.

Usually the code fragments are about 1 kB or smaller. If enough code is executed between two repetitions of our hypothetic fragment to push it out of several kB of instruction cache, it has to take several times longer. Worst case scenarios can be constructed, but the empirical results [3] show that they are not likely to happen. Altogether, we conclude that for each code fragment there is a constant C_j such that the context-sensitive execution time of the fragment lies in some interval $[C_j/\bar{F}, C_j\bar{F}]$ for some hardware dependent $\bar{F} << \delta$.

The approximation neglects pipelining. The experiments in [3] show that the speedup resulting from a pipeline depends strongly on the optimization of the compiler and the individual pipeline. There are three classes of events that decrease the efficiency of a pipeline by producing a 'stall' [3]:

1. A resource conflict of the hardware. An instruction pair can not be executed with overlap.

2. A data conflict, if an instruction needs the result of a previous instruction.

3. Instructions that change the program counter, as branches and jumps.

The proposed method of equivalent code fragments times code fragments that are very similar to the actual code fragments. Thus approximately the same number of stalls is to be expected and therefore pipelining has very little influence on the quality of our predictions.

4 Methods

A first idea about the execution time of an algorithm can be obtained by simply running it on a few different inputs. Together with the theoretical analysis this simple approach gives the order of magnitude of the expected execution time in many cases. But no claim about the accuracy of the determined values can be made and automation requires generators for feasible inputs and a more sophisticated analysis, which leads us back to the determination of a set of running time constants. Two approaches to determine the constants C_i are described in the literature.

4.1 Regression Methods

The first class of methods is the numerical analysis of experiments. It is for example used in [6] and [8]. 'Counters' at feasible positions in the program provide the execution numbers for individual inputs. Even for complex algorithms the number of necessary counters is limited. In [6] the authors represent code subsets with so called 'bottleneck operations'. Even if not only the leading terms are taken into account, the number of necessary counters is small. For Dijkstra's algorithm based on a Fibonacci heap 9 counters are sufficient.

The result of a set of N measurements with different inputs is a set of N data points $(t_i, f_i^{(1)} \dots f_i^{(M)})$, $i = 1, \dots, N$ where t_i is the measured running time and $f_i^{(l)}$ is the execution frequency of the l-th code fragment. We also have a functional relationship

$$T = \sum_{j=1}^{M} f^{(j)} C_j (1 + \varepsilon_j)$$

with unknown constants C_i and small ε_j's depending on the context of the j-th execution. Fit methods determine constants $\alpha_1, \dots, \alpha_M$ such that the model function $T' = \sum f^{(j)}\alpha_j$ approximately passes through the given data points. There is no reason to believe that this implies that the α_j's approximate the C_j's. In fact we saw negative values for α_j's in experiments (which used singular values decomposition [5] for the fit.

4.1.1 Least Square Approximation

Least square approximation is a popular fit method. It assumes that the deviation

$$\Delta_i = t_i - \sum_j f_i^{(j)} C_j$$

in the i-th measurement is normally distributed with some standard deviation σ (independent of i) and that the errors in the measurements are independent. Under this assumption the probability P for a given set of N measurements (assuming the α_i to be correct) is given by

$$P = \prod_{i=1}^{N} \left(\exp\left[-\frac{1}{2} \left(\frac{t_i - t(f_i^{(1)}, \dots, f_i^{(M)})}{\sigma} \right)^2 \right] \Delta t \right)$$

Least square approximation determines the parameters $\alpha_1, \dots, \alpha_M$ so as to maximize P.

In our case the deviation Δ_i consists of two parts, namely the modeling error $\sum f_i^{(j)} C_j \varepsilon_j$ and the measurement error (the difference between the true running time and the measured running time). Only the measurement error is statistical, the modeling error is systematic.

4.1.2 Systematic Errors

For the time measurements described in section 3, the dominating error is caused by the context sensitivity of the execution time of an instruction, since the model does not take features as caching and pipelining into account. For example, an input with high locality is processed significantly faster than an input with lower locality. This is a systematic error.

How does this systematic error influence the fit of a running time function? The systematic error defines a qualitative behavior that is approximated by the fit. Assume a model function

$$m(N) = A + B \log N + CN$$

and a set of measurements of $m(N)$ for different N, where N is the input size and a smaller N is equivalent to a higher locality in the program. A cache causes a positive second derivative of $m(N)$. The fit approximates this positive second derivative by increasing the value of C and by making B negative. The relative error of B is bigger than 1. Additionally, the leading coefficient is increased by an unknown factor that depends on the choice of the input sizes.

Fit methods are extremely sensitive against systematic errors as they appear in the running time measurements. Even if a set of experiments is chosen 'well', bounding the errors in the coefficients is impossible, since the conditions for the statistical analysis, correctness of the model function and a known error distribution, are not fulfilled.

4.2 Operation Counting

The second class of methods basically counts all operators, function calls and references in a program. Since these instructions can be mapped to assembler code, the expected number of clock cycles for the execution is known and on this base a time constant can be calculated. This method gives feasible results [2] but it is difficult to automate.

One approach for automation is the modification of a compiler. The compiler just counts the weighted operators (function calls are represented by the () operator) in a loop or function for subsets of code identified by the user. But this solution depends on the compiler which has to be available on all different hardware platforms and for the different programming languages. Additionally the calculation of the context dependent weights and the modification of a compiler is costly.

Operator overloading provides a second way to count operators automatically. The operations of a program are replaced with versions, which count themselves depending on the context. But in this case a data structure has to be maintained that tracks the context during the execution of the program. The implementation of this data structure is basically as complex as the investigated algorithm itself. Additionally, this approach depends strongly on the language and requires usually changes in the implementation of the algorithm. In C++ the whole class of pointer declarations can not be overloaded. For example each declaration of an array *type a[]* and each access *a[i]* has to be replaced in the implementation by defining an appropriate class.

A profiler determines directly the running times for code fragments. If the profiler provides the execution time for each line of code, the running time constants can be determined out of these data. But the profiler requires additional code in the program and code optimization is impossible or disturbs the measurement of the profiler. The profiler gives information about the relations between the running times of a program, but not the absolute values.

As a result, operation counting provides feasible values for the constants, but the automation is costly and depends on the language or the compiler.

4.3 Equivalent Code Fragments

Since fitting data for different inputs is not sufficient to control the error in the time prediction, a different method is necessary to determine the constants C_i. The following properties are required:

1. The constants C_i are determined within a constant factor, that depends only on the hardware. It is sufficient to control the error in the single coefficients, since the running time is a linear function in these coefficients.

2. The results are compatible with modular or object oriented programming. If an algorithm is used as a subroutine, its constants can be reused. Only the new code has to be investigated.

3. No changes in existing code are necessary. For the automatic determination of the constants only an additional set of functions, or methods from the object oriented point of view, is necessary. The concept is independent of the programming language in a certain range (C, C++, Pascal, assembler, ...). Programming environments that include the execution of indirect tasks, like automatic garbage collection, are not allowed. Such tasks are separate algorithms which have to be analyzed separately.

The concept of *equivalent code fragments* provides these properties. As we mentioned, even complex algorithms consist of a limited number of subsets of code (code fragments), executing each fragment with a certain frequency. The target is the approximation of the running time of the individual code fragments that are the constants C_i in the model function.

Since the separation of individual constants by a fit is not feasible, the constants are determined with individually designed experiments. However, real problem instances and the original code are not used, they are represented by modified code fragments and special inputs for these fragments. The modified code fragments have to be similar in the number of operations and the locality to the original code fragments, they are *equivalent*.

As an example we present here one of the equivalent code fragments for the investigated implementation of Dijkstra's algorithm, the interior of the loop which cuts nodes out of the heap during a DecreaseKey-operation. MFHeapCut is a function with code, which is executed once per call. The code fragment is embedded in a loop performing the repetitions to achieve sufficiently long execution times.

```
InitMeasure();
for (j=0;j<r;j++) {
    x = parent;
    do {
        y = x->parent;
        MFHeapCut(H,x);
        if (x->key < H->min_ptr->key)
            H->min_ptr = x;
            x=y;
    } while(x->mark);
}
EndMeasure();
blackhole(1, &( H->min_ptr->key));
```

Two problems have to be considered in the determination of the constants. The first is the design of feasible equivalent code fragments for the individual constants. The second is the automation of measurements and their analysis. If a measurement is not successful this has to be recognized instead of providing a wrong constant if possible.

Goal of the experiments is the determination of the running time for the execution of a code fragment of the implementation. From the analysis of the possible errors follow some rules for the design of equivalent code fragments.

• The running time of the experiments depends on the amount of memory that is used, since this influences the locality of the references. An experiment should use a similar amount of memory as the original code. In the memory range, that requires no swapping, the running time approaches asymptotically a maximum. As a result a good extrapolation behavior can be expected.

• To guarantee a sufficient accuracy of the time measurement, 'minimum measurement time loops' (MMTL) should be used wherever possible. The experiment measures the time for a number of repetitions of the code fragment. This number is increased if the total time is smaller than some constant. All tested systems provide a timer with an accuracy of at least 50 ms, so a minimum time of 2000 ms guarantees a sufficient accuracy.

• The measured code should execute at least 5 times more instructions than the MMTL environment.

• The system calls *malloc* and *free* have to be handled with care. They do not have a constant running time per call on all systems. They are separate algorithms, that have to be analyzed separately. The SunOS version of *free* has a worst case running time that is linear in the number of earlier allocated blocks. The Linux version does not show this effect. But on the tested platforms the assumption of constant execution time for these functions is feasible.

• Experiments have to be designed in a way, so that an optimizer isn't able to remove repetitions. A function *blackhole* can be used which accepts a pointer as an argument and is compiled in its own module. Calling this function with a data structure 'by reference' outside of the measurement loop is a helpful tool for this task.

4.3.1 Automation

The determination of the running time constants is performed automatically by two additional programs, the *controller* and the *worker*. They perform the individual experiments and calculate the execution time constants. We discuss some design issues for the controller and the worker.

• Cumulative memory fragmentation from one experiment to the next has to be avoided. For this purpose, the experiments are combined in one program (the *worker*), which performs one of them per call, controlled by command line parameters. The management of the experiments is done by a second process (the *controller*), which starts the *worker* with the Unix system call 'system()' or an equivalent system function on other systems. Due to this structure each experiment starts with a freshly initialized internal memory management of the *worker*.

• To consider different cache hierarchies and timing accuracies on the target platforms, the experiments accept parameters to control the number of repetitions, the minimum execution time and the amount of elements (memory) for the execution. As a result the dependency of constants on memory usage can be investigated.

• During the experiments, only the necessary system processes are allowed to run, since the elapsed time is measured, not the CPU time. CPU time does not consider minor and major page faults and time spent with waiting for data from hard-disk or network [9]. Programs which use dynamic storage allocation cause minor page faults even if no swapping is necessary. CPU time can be used as an approximation but then only CPU time is predicted, not elapsed time.

• If not enough free memory is available for the algorithm, inactive code is swapped or rearranged to obtain sufficient connected memory. Repeated execution of the experiment and careful evaluation of the data insures, that the calculated constants are not disturbed by this effects. As a result, the execution of the algorithm in a program might take a small and constant amount of time longer than the predicted time, depending on the total amount of memory the system has to provide for it. But for bigger instances which need more than a few seconds this error is neglectable. The times for real problem instances given in figure 3 are measured executing the algorithm once, but with sufficient free memory for the process.

• Obviously the predicted running times apply only if there is sufficient memory to keep algorithm and data in main memory during execution.

Experiments were performed on the following hardware platforms:

1. Pentium 133 MHz, 32 MB RAM (60 ns), 8 kB internal cache for instructions and 8 kB for data, 256 kB external pipeline-burst-cache, operating system Linux.

2. Sparc ELC, 16 MB RAM, operating system SunOS.

3. Sparc 5, 85 MHz, 64 MB RAM, 16 kB internal cache for instructions and 8 kB internal cache for data, operating system Solaris.

4.Sparc 4, 110 MHz, 64 MB RAM, internal cache, 16 kB for instructions and 8 kB for data, operating system Solaris.

Since only a few basic system calls are used, the sources, makefiles and scripts could be used on all platforms without change. Only a possibility for time measurement and the execution of processes out of another process are necessary, so the transfer to non-Unix systems is not costly.

4.3.2 Analysis of the Measurement Results

Each experiment provides one (single run) or a sequence of values (MMTL) for a constant. For the time measurement the function *gettimeofday()* is used. The evaluation of the measurements is non-trivial. There are systematic errors and outliers. Outliers, which are much higher than the other values, occurred in many of our experiments. They occur especially on multitasking systems, but not only on them. Even if no other user processes are running, some interruptions by the system are possible. Another typical error is observed in the results of MMTLs. The short total time of a small number of repetitions produces an error due to the accuracy of the time measurement. For a high number of repetitions the memory management causes an increase of the values even before the system reports a major page fault in the result of a *getrusage()* call, due to minor page faults.

To eliminate outliers before calculating the average, a robust method is necessary for the evaluation. Fig. 1 shows a sequence from a MMTL run, which contains several outliers.

If no other processes disturb the measurements, a cumulation of values can be expected. This property is used to eliminate strongly defective values. Let M be a set of N measurements. We choose a constant $\Delta > 1$

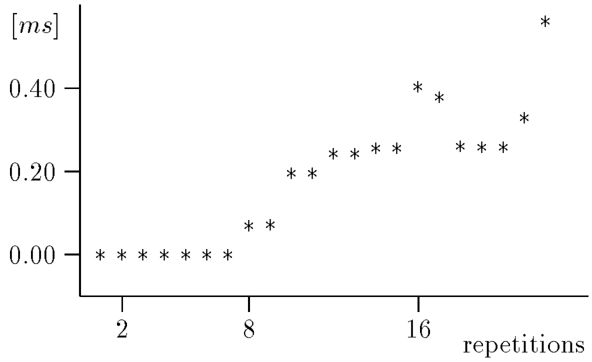

Figure 1: Sequence of values from a MMTL with 1, 2, 4, 8, ... repetitions.

(we used $\Delta = 10$) and search for an interval

$$[R - \frac{R}{\Delta},\ R] \text{ with } \#([R - \frac{R}{\Delta}] \cap M) > \frac{1}{2}N\,,$$

i.e. a short interval containing the majority of the data points. All data points outside the interval $[R - R/\Delta]$ are considered outliers.

The measurement values are sorted by their value and the algorithm starts with an interval $I_r = [R - R/\Delta, R]$ where R is the biggest value. Now smaller measurement values are chosen in decreasing order for R as long as I_r contains less than $N/2$ values. If no interval is found the set of measurements is not accepted, otherwise the average of the values in the interval with lowest R found is returned as approximation of the constant.

5 Example Algorithms

As mentioned above we have chosen a hierarchy of algorithms to test our method.The top layer is a MWBM-algorithm which determines the heaviest matching in a weighted bipartite graph (Contrary to the assignment problem, the maximum matching does not have to be perfect).

The MWBM-implementation uses a modified version of Dijkstra's algorithm. The modification is an additional condition which stops execution if a feasible augmenting path is found. The original implementation of Dijkstra's algorithm as well as the modified version both use the implementation of a Fibonacci heap as priority queue and a data structure GRAPH based on adjacency lists. Due to the similarity the modified version of Dijkstra's algorithm is assumed to have the same execution time constants as the original.

Running time predictions for these algorithms is complicated by the following facts: Only an amortized analysis of the costs is possible and the execution frequency of the code fragments depends strongly on the choice of the edge weights and the order of the edges in the adjacency lists, not only on the size of the input graph G.

Dijkstra's algorithm needs $O(N + M)$ steps of constant time and N INSERT, N DELETEMIN and $O(M)$ DECREASEKEY operations on the priority queue for a graph with N nodes and M edges. Let $G_b = (V_a, V_b, M)$ be a bipartite graph, and V_a the smaller set of nodes without loss of generality. Our implementation of the weighted bipartite matching algorithm starts Dijkstra's algorithm for each node $x \in V_a$ to determine a feasible augmenting path starting at x, augments along this path by reversing the edges and updates the node potentials. $N_a = |V_a|$ calls of Dijkstra's algorithm, N_a augmentations along paths with a length of at most $2N_a$ and N_a updates of at most $N_a + N_b$ node potentials are performed.

5.1 Running Time

Although a detailed description of the implementations and the resulting constants exceeds this paper, we give expressions for the different running times as examples for possible types of running time predictions.

5.1.1 Fibonacci Heap

The basic operations *Insert*, *DeleteMin*, *DecreaseKey*, *CreateHeap* and *DestroyHeap* can be expressed in terms of 8 counters and 12 constants. Each execution time is an equation of the type (1.1). The set of constants is determined by 13 experiments.

The additional operation *DestroyHeap* deallocates the elements in the heap by traversing the data structure recursively, which is more efficient than repeated calls of *DeleteMin*. This allows an improvement in the MWBM-implementation, since the modified version of Dijkstra's algorithm used by the MWBM-algorithm does not remove all elements from the heap.

The worst case analysis provides an upper bound for each counter in the execution time. Replacing the counters with this bounds gives a worst case approximation of the running time. For a sequence of N *Insert*-, L *DeleteMin*- and M *DecreaseKey*-operations we have

$$\begin{aligned} T_{heap}(N, L, M) \quad &\leq \quad C_{H1} + (N - L)\, C_{H2} \\ &\quad + N\, C_{H3} + M\, C_{H4} + L\, C_{H6} \\ &\quad + 1.5 L \log_2(N)\, C_{H5} \end{aligned}$$

The constants C_{Hx} in this expression are sums of subsets of the 13 values determined experimentally. If each *DecreaseKey* violates the heap conditions, this prediction is tight within a small constant factor.

5.1.2 Dijkstra's Algorithm

The execution time for Dijkstra's algorithm involves four additional constants.

$$\begin{aligned} T_{Dijk}() \quad &= \quad N_{D1}\, C_{D1} + N_{D2}\, C_{D2} \\ &\quad + N_{D3}\, C_{D3} + \sum_{j=1}^{N_{DecKey}} T_{DecKey}(j) \\ &\quad + C_{D4} + T_{Destroy}() + \sum_{j=1}^{N_{D3}} T_{DelMin}(j) \end{aligned}$$

With upper bounds for the counters N_x the worst case execution time is approximated by

$$\begin{aligned} T_{Dijk}(N, M) \quad &\leq \quad M\, C_{D1} + T_{heap}(N, N, M) \\ &\quad + C_{D4} + N\, (C_{D2} + C_{D3}) \end{aligned}$$

for an input graph with N nodes and M edges.

For random connected graphs the execution time can be approximated by

$$\begin{aligned} T_{Dijk}(N, M) &\approx C_{D4} + C_{H1} + M\, C_{D1} \\ &\quad + N\, (K_{D2} + K_{D3} + C_{H3} + C_{H6}) \\ &\quad + 1.5\, N \log_2(N)\, C_{H5} \\ &\quad + (N \log_2(1 + M/N) - N)\, 0.5\, C_{H4} \end{aligned}$$

since it is unlikely that the shortest paths contain a large number of edges [11] [12].

The execution time for the maximum bipartite matching algorithm is

$$\begin{aligned} T_{Mwbm}() \quad &= \quad \sum_{i=1}^{N_A} [T_{Dijk}(i) + T_{MAug}(i) + T_{MUpd}(i)] \\ &\quad\quad + C_{Mwbm} \\ T_{Aug}(i) \quad &= \quad C_{Aug} + N_{Aug}(i)\, C_{AugL} \\ T_{Upd}(i) \quad &= \quad C_{Upd} + N_{Upd}(i)\, C_{UpdL} \end{aligned}$$

The code of the main loop can be neglected compared to the function calls. In the worst case, the execution time is

$$\begin{aligned} &T_{Mwbm}(N_a, N_b, M) = \\ &\quad C_{Mwbm} + [N_A + N_B]\, C_{MInitL} + N_A\, C_{MwbmL} \\ &\quad + N_a\, [T_{Dijk}(N, N, M) + T_{Aug}(N_a) + T_{Upd}(N)] \end{aligned}$$

5.1.3 A Code Fragment - CutLoop

In this section the experiments for the code fragment listed in section 4.3 are described as an example. The code fragment corresponding to the constant $C_{CutLoop}$ is performed during the DECREASEKEY-operation. Two experiments were implemented to determine $C_{CutLoop}$ to get the possibility to compare the

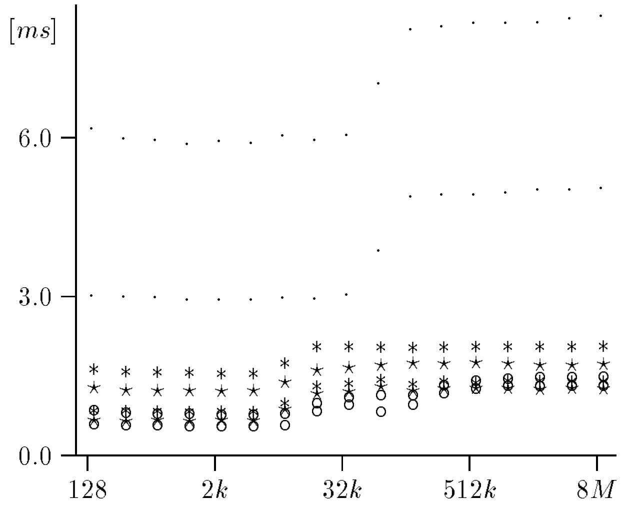

Figure 2: The dependency of $C_{CutLoop}$ on memory usage, with and without optimization. (. = ELC ; $*$ = Sparc 5 ; $\star$ = Sparc 4 ; $\circ$ = Pentium 133) .

results of a direct loop on the original code and a modified code fragment in a MMTL. Both experiments use the same special heap structure that results in N executions of the loop.

The difference of the 2 experiments is the function call to cut a node. The first version of the experiment uses the original function FHEAPCUT, the second version uses a modified version MFHEAPCUT that performs almost the same operations, but without changing the data structure. This means the nodes are not really cut. Since the first experiment changes the data structure, no MMTL is possible in this case.

The amount of memory usage is 64 bytes per element. The cache structure of the different platforms is visible in the experiments (Figure 2). The direct experiments with 100000 elements give similar values to the values in figure 2 (**P133** -g: 0.951 μs , -O: 0.811 μs; **Sparc 5** -g: 1.9 μs , -O: 1.25 μs; **Sparc ELC** -g: 7.6 μs , -O: 4.6μs).

5.1.4 Time Prediction

The running time constants were automatically determined with identical code on the different hardware platforms. Constants for code fragments with fixed memory usage were calculated as the average of 10 measurements. The constants for scalable experiments were determined out the set of values given by the MMTL. First the smallest value is identified. This value belongs to an experiment with low memory usage. As a second value the biggest value out of the experiments with

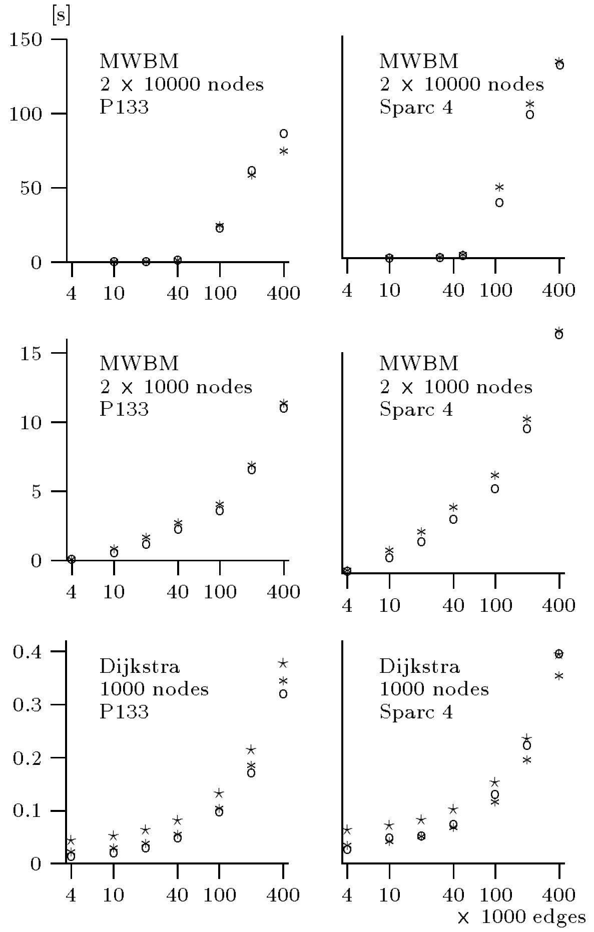

Figure 3: Execution times ($*$ = predicted with frequencies ; $\star$ = average case analysis; $\circ$ = measured), compiled with 'gcc -O'.

higher memory usage than the first identified experiment is chosen. This prevents taking errors into account due to the MMTL environment code which increases the constants for very low memory usage in some cases.

Although with $\Delta = 10$ a strong stability criteria was used during the elimination of strongly defective data points, most experiments were successful. Only a few values for a certain size of an experiment were not determined in the first run. But in this case sufficient values for other sizes were obtained to calculate the median out of the values for different sizes. An experiment is called successful if the point elimination terminated. The algorithm stops without result if no intervals with more than 50 % of the data points are found.

If an experiment is not successful, an automatic 3 step strategy is possible to get a result. In many cases a simple repetition of the experiment is successful, if the drop out was caused by a temporary interruption. If the experiment used an MMTL, the rate of the exponential increase of the repetitions can be decreased. So more points in the range with high accuracy are obtained. As a last step the parameter Δ can be decreased.

6 Conclusion

As the experimental results show, the concept of equivalent code fragments provides a method for the automatic determination of running time constants within a small constant factor. Even without the investigation of the memory dependency of the constants good approximations can be expected by choosing experiments with a memory usage in the order of magnitude of the size of the cache.

Only basic system functions are used, which are available in most programming environments. Since the experiments are supposed to be similar to the original code fragments, their design and implementation is less costly than the implementation of the algorithm itself. The calculated coefficients are compatible with the concepts of modular and object oriented programming when the algorithm is used in a wider context.

The comparison with operation counting shows, that the method of equivalent code fragments is less costly to automate. Additionally the context of the executions is considered partly due to the similarity between code and experiments, which is very difficult in the case of pure operation counting.

References

[1] Donald E.Knuth, *The Art of Computer Programming.* Addison-Wesley (1968)

[2] D.E. Knuth. The Stanford Graph Base. *Addison Wesley, New York* (1994)

[3] J.L. Hennessy, D.A. Patterson. Computer Architecture - A Quantitative Approach. *Morgan Kaufmann Publishers, Inc., California* (1990)

[4] T.H. Cormen, C.E. Leiserson, R.L. Introduction to Algorithms. *MIT Press, Cambridge, Massachusetts* (1990)

[5] W.H. Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling. Numerical Recipes in C. *Cambridge University Press* (1988)

[6] R.K. Ahuja, T.L. Magnanti, J.B. Orlin. Network Flows: Theory, Algorithms, Applications. *Prentice Hall, Englewood Cliffs, NJ* (1993)

[7] A.V. Aho, J.E. Hopcraft, J.D. Ullman. Data Structures and Algorithms. *Addison Wesley, Reading, Mass.* (1983)

[8] R.G. Bland, D.L. Jensen. On the computational behavior of a polynomial-time network flow algorithm. *Mathematical Programming*, **54** (1992) 1-39

[9] Kevin Dowd. High Performance Computing. *O'Reilly & Associates, Inc., 103 Morris Street, Sebastopol, CA 95472* (1993)

[10] Kurt Mehlhorn, Stefan Näher. *LEDA, a Platform for Combinatorial and Geometric Computing.* Communications of the ACM, volume **38**, (1995), pp. 96-102

[11] Koshei Noshita. *A Theorem on the Expected Complexity of Dijkstra's Shortest Path Algorithm.* J. Algorithms **6**, (1985), pp. 400-408

[12] A,V, Goldberg, R.E. Tarjan. *Expected Performance of Dijkstra's Shortest Path Algorithm.* NEC Research Institute Report 96-062.