FAKULTÄT FÜR !INFORMATIK

A Simplex-Based Heuristic for Efficient Workflow Composition

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Alexander Körpert
Matrikelnummer 0225878

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Univ.-Prof. Dr. Schahram Dustdar
Mitwirkung: Univ.-Ass. Dipl.-Ing. Martin Vasko

Wien, 02.12.2008

(Unterschrift Verfasser)

(Unterschrift Betreuer)

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wien, 2. Dezember 2008

Acknowledgements

I want to express my gratitude to Professor Schahram Dustdar, for giving me the opportunity to write my thesis on this interesting subject at his institute.

I also thank Martin Vasko for his valuable advice, constant engagement and feedback. Besides, I appreciate his organisational help, especially at the completion of this work.

Special thanks go to my parents Franziska and Karl Körpert for supporting my studies throughout the years.

Abstract

Productivity at large scale raises the demand for highly automatic and dynamic activity scheduling mechanisms. No matter if the task conforms to modelling the manufacturing process of a car or the offering of an electronic service querying multiple web services on a certain user request. Central questions such as “Which activity fits best for this task?” determine the composition of workflows before and during runtime.

This work investigates a simplified abstract workflow model whose mathematical representation corresponds to a binary integer linear program. The evaluation criteria of activities is based on a single benefit and cost value. Selecting those activities with maximum overall benefits and total costs lower than a predefined limit leads to an optimised concrete workflow.

A heuristic based on the revised simplex method takes advantage of rounding the non-integral solution. The algorithm together with an on-the-fly read-in procedure was implemented as C library. Furthermore, an OO-Wrapper steers execution for C# applications. The test project comprises random tests for verifying functionality and facilitating runtime experiments.

The algorithm solves large problem instances involving about 100 000 variables in 1.3 seconds on a 2.0 GHz Dual-Core processor. The relative error is guaranteed not to exceed the reciprocal of the number of nodes per route. Configurable use of memory resources renders the application also attractive for devices with limited storage capabilities. Concerning these performance features, an integration into a workflow framework sounds very promising.

Zusammenfassung

Produktivität in Geschäftsprozessen erfordert hoch automatisierte und dynamische Mechanismen zur zeitlichen Einteilung von Aktivitäten. Dabei spielt es keine Rolle, ob die zu erledigende Aufgabe der Modellierung eines Herstellungsprozesses von Autos oder der Beschreibung eines elektronischen Services, welches Anfragen an unzählige weitere Web Services weiterleitet, entspricht. Zentrale Fragen, wie „Welche Aktivität ist für diese Aufgabe am besten geeignet?“ bestimmen die Zusammenstellung von Workflows vor und während der Laufzeit.

Diese Arbeit untersucht ein vereinfachtes Modell eines abstrakten Workflows, welches den Sachverhalt als ganzzahliges 0/1-Programm formuliert. Das Bewertungskriterium von Aktivitäten fußt auf der Zuordnung eines Kosten- und Nutzenwertes. Die Auswahl jener Aktivitäten mit maximalem Gesamtnutzenwert bei Gesamtkosten, die einen vorgegebenen Grenzwert nicht übersteigen, führt zu einem optimalen konkreten Workflow.

Eine Heuristik, die auf dem revidierten Simplex-Verfahren basiert, macht sich das Runden der im Allgemeinen nicht ganzzahligen Lösung zu Nutze. Der Algorithmus wurde zusammen mit einem „On-The-Fly-Einlesevorgang“ als C Bibliothek implementiert. Weiters steuert für C# Applikationen ein objektorientierter Wrapper den Zugriff auf die Bibliotheksfunktionen. Das Test-Projekt umfasst Zufallstests für die Verifizierung der Funktionalität und ermöglicht darüber hinaus Experimente zur Laufzeitbestimmung.

Der Algorithmus löst große Probleminstanzen mit ca. 100 000 Variablen auf einem 2.0 GHz Dual-Core Prozessor in 1,3 Sekunden. Dabei übersteigt der relative Fehler jedoch nicht den Reziprokwert der Knotenzahl pro Route. Der konfigurierbare Speicherverbrauch macht die Applikation auch für Geräte mit begrenztem Speicher interessant. Diesen Leistungsmerkmalen zufolge klingt eine Einbindung in ein Workflow Framework äußerst vielversprechend.

Contents

List of Figures

List of Tables

Code Listings

1 Introduction

Optimising workflows is always desirable if a surplus of possible constituents exists. The elements of a business process are commonly named activities or services carried out by humans or computer applications. Generally, they differ in the quality of service (QoS) criteria such as availability, reliability and execution time. This quantifiable measure characterises not only a single service but also an arbitrary aggregation of activities. A comparison quickly points out an optimal configuration – the process of finding the best composition may take very long by contrast.

1.1 Motivation

Workflows often involve a high number of activities. This results from the fact that operating processes tend to be large and can be splitted into many parts in addition. If for instance, the software engineering process has to be modelled as workflow then the size depends on the chosen granularity i.e. the level of detail the activities stand for. Repetitive operations on business processes also contribute to a high net-length. In particular, the services within a loop will have to be multiplied by the number of iterations.

Eventually, a group of activities may have the same functionality necessary to accomplish a task. In that case, a representative suiting best according to its QoS criteria can be chosen. If the selection happens to be independent from previous ones, the problem will reduce to a local decision on the actually best fitting candidate. However, most often the composition takes place under global constraints. It is e.g. convenient to fix at least an upper bound for a workflow's total execution time while maximising the other properties. Likewise, lower limits for the overall availability and reliability to which a solution of a minimal execution time has to adhere, can be set. Global constraints of that kind usually render the problem computationally difficult, especially in case of functionally equivalent service providers.

The demand for dynamic updates constitute a third characteristic inherent to mainly large business processes: QoS criteria may change during their execution, new activities can get inserted while others are maybe deleted. Besides static optimisation, an evaluation at runtime also needs to be facilitated by the responsible software component.

1.2 Challenge

Addressing combinatorial aspects and dynamic issues for potentially large workflows, represents the main goal of their automatic composition. Assigning the selection process to the lowest architectural tier enables a transformation of the original workflow. The latter e.g. serves for sequentialising loops (to get the net-number of activities) and determining an execution order (for synchronised services). Moreover, an appropriate abstraction of an activity in the original workflow has to be found as well. It should provide only the data, the evaluation is performed on.

Every graphical representation corresponds to a mathematical formulation and vice versa. The optimisation problem can be modelled in different ways as outlined in chapter 2. The quality of the solution and the time for its detection will often emerge as opponents. The latter plays an important role, especially if a dynamic update is inevitable: Depending on the position where execution has stopped, a subgraph of arbitrary size should be re-evaluated as fast as possible.

1.3 Overview

The remainder of this work is organised as follows: In chapter “Related Work”, the state of the art of workflow optimisation is highlighted. The main part focuses on the analysis and comparison of approaches similar to the one taken in chapter 3. The latter introduces an abstract workflow model with a binary linear program as mathematical formulation. For the solution finding process, a simplex based heuristic is suggested with a glimpse at exact methods. The next chapter displays the realisation of preceding concepts. Furthermore, section "Usage" shows relevant setup information and useful hints at a glance for the busy reader. In "Evaluation", the library is examined under three aspects: exactness, memory and runtime. Based thereon, two different customisation scenarios are presented. The last chapter "Conclusion" contains some final remarks, lists some open points for a future release and finally provides a summary of the work.

2 Related Work

In literature, the term “workflow optimisation” goes hand in hand with a QoS based optimal composition of web or grid services. The mathematical formulations as optimisation problems vary greatly in the number of resources and constraints. With respect to the approach taken in this work, the focus lies in analysing and comparing multiple choice knapsack models used by several authors.

2.1 State Of The Art

The QoS evaluation of web services and the thereon based optimal selection have grown in interest over the last few years. Different algorithms have been developed which can be roughly divided into exact, approximative and heuristic ones. Recently, it has been shown by Yu et al. [YZL07] that an optimal solution to the integer linear program can not be found in reasonable time for very large instances. Theoretically, this is explained as a consequence of the NP-hardness inherent to these problems. Heuristics, on the other hand, usually perform better (at the disadvantage of accuracy though) as demonstrated e.g. by Yang et al. [YTX+07].

2.2 Literature

Yu et al. [YZL07] designed a broker-based architecture for service detection, planning, selection and adaptation. The first describes the lookup of available services in consideration of QoS aspects. A process plan determines component functions and their dependencies. The binary multidimensional multiple choice knapsack model represents the combinatorial model taken in step 3. Hence, the objective is to maximise a certain utility function subject to QoS constraints such as response time, cost and availability. For that purpose, they compared a branch-and-bound algorithm based on linear programming relaxations with a heuristic algorithm that finds a statistically good solution in polynomial time. The adaptation of a service denotes an update of its actual QoS values.

Yang et al. [YTX+07] investigate service composition under

1. no global constraint,
2. a single global constraint and
3. multiple global constraints.

The first problem reduces to a local decision on a best matching candidate that requires only linear time. The second and third assignment have been identified as multiple choice binary knapsacks (differing in dimension). A heuristic which determines the convex-hull in $O(nlog(n))$ has been used in both cases. Besides, the excellent scaling behaviour (as expected due to the low complexity), the experiments confirmed a satisfying quality in average.

Menascé et al. [MCD08] formulate a non-linear optimisation problem where a solution minimises the average execution time of a business process. The latter together with the total costs of execution have to be computed from the workflow representation (i.e. BPEL in this case) first. The developed exact algorithm is suited well for small or moderate instances. The larger ones are preferably solved with the additionally provided heuristic whose experimental results are close to 6% of the optimum.

A mixed-integer linear programming formulation is explored by Anselmi et al. [AAC07] within their QoS broker based framework for web services in autonomic grid environments. Combining several requests for a simultaneous evaluation renders the task to a multiple instance web service composition problem. A greedy heuristic is applied to its solution. Experiments classified the quality of the output as “good” and the reduction of the computation overhead compared to existing techniques as significant.

Canfora et al. [CDPEV05] propose a generic algorithm for the QoS based (web) service selection. More specific, a decision procedure decides on a set of concrete services to be bound at runtime to abstract ones, taking into account various constraints and QoS attributes. According to the authors, this approach is slower than using linear integer programming but more appropriate for generic QoS criteria. Furthermore, the aggregation functions can be non-linear, as well.

3 Approach

In this chapter a new approach which meets the given requirements is presented. However, this model is applied to the composition and evaluation of workflows at the lowest architectural tier. Therefore, some general assumptions according to its application are necessary.

First, workflow patterns as described in Jaeger et al. [JRGM04] need to be primarily handled by an upper layer. Figure 1 shows the decreasing levels of abstraction for an example business process. The model itself is a graph where loops have been sequentialised and an execution order for parallel splits and synchronisation has already been determined.

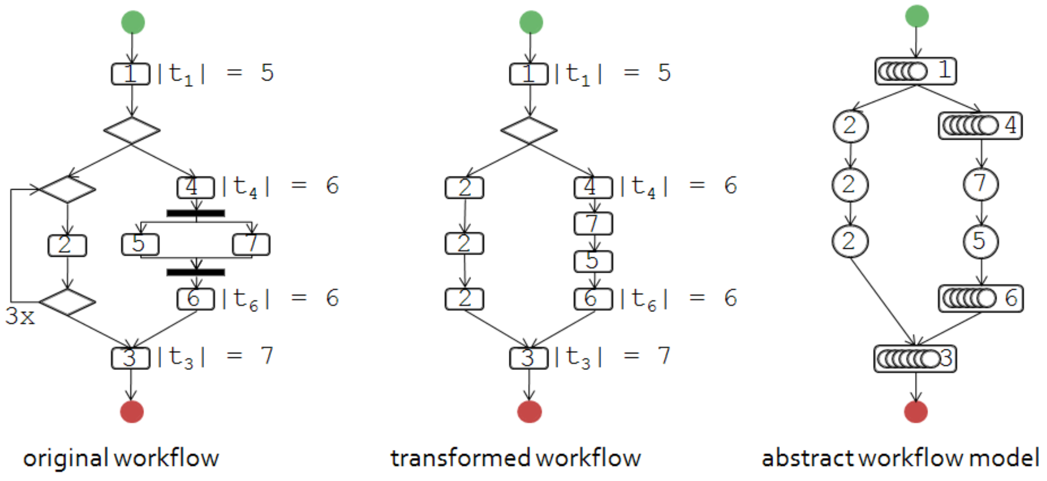


Figure 1: Representations of an example workflow. The abstract workflow model corresponds to the lowest architectural tier.

Second, an activity itself is simplified as much as possible. It constitutes an abstract representation of a real world service (e.g. a web service or human interaction). Criteria for activity selection are usually based upon characteristics such as execution time ($t_e[s]$), reliability (r) and availability (a) combined to a single benefit (b) and cost value (c) by the calling application. In common, the mapping takes the form:

$$b: \quad \mathbb{R} \rightarrow \{b \mid b_{min} \leq b \leq 2b_{min}, \quad b, b_{min} \in \mathbb{N}_0\} \tag{3.1}$$

$$c: \quad \mathbb{R} \rightarrow \{c \mid c \leq C_{max}, \quad c, C_{max} \in \mathbb{N}_0\} \tag{3.2}$$

Additionally, $\mathbb{R}$ and $\mathbb{N}_0$ have to be replaced by the appropriate number range. As the four byte "int" has been chosen as variable type for b and c, $\mathbb{N}_0$ is substituted by the interval $[0, 2^{31} - 1]$. A further constraint concerning the codomains of b and c results from the fact that the sums of benefits and costs have also to be within $[0, 2^{31} - 1]$: For example, 10^7 nodes suggest $b_{min} = 100$ and an average maximum cost value of 200 in order to prevent an internal overflow.

As already indicated, ignoring any of these restrictions may lead to invalid or unintended results. Regarding the latter, trying to assign $b < b_{min}$ or $b > 2b_{min}$ corresponds to $b = b_{min}$ or $b = 2b_{min}$ respectively. By the way, the reason for narrowing b to $[b_{min}, 2b_{min}]$ lies in the provable upper bound of the relative error due to this restriction (shown in section 5.1). The following functions for instance provide a simple and correct mapping:

$$b(a,r) = \lfloor 100(1 + ar) + 0.5 \rfloor \tag{3.3}$$

$$c(t_e) = \lfloor 100 t_e / t_{exp} + 0.5 \rfloor \tag{3.4}$$

The variable t_{exp} refers to the time (in seconds) a task is expected to need. The values for a, r, t_e and t_{exp} have to be measured or estimated in advance. If they change during runtime, the library can be invoked with the remaining subtree of altered activities. Generally, it's the low execution time that enables dynamic updates.

3.1 Abstract Workflow Model

An abstract workflow comprises all execution routes between initialisation and termination. The graph is acyclic and a node may have more than one predecessor (as opposed to a tree). Anyway, the in-order processing works too, although the read-in might get inefficient if all successors of such kind of node have to be inserted again.

As illustrated in figure 1 a task conforms to a node in this model. It complies with a single or pool activity. Pools with fewer than two candidates are accepted as well. However, this usage is discouraged because they unnecessarily slow down execution and also come in limited quantities.

As indicated above, each activity holds a benefit/cost value pair. Based on these evaluation criteria, the composition of a concrete workflow through an efficient and optimal selection process (i.e. maximising the sum of benefits subject to constraints) constitutes the main challenge. Concerning mathematical side conditions, the total costs of a business process instance must not exceed the predefined limit C_{max} (e.g. a mapping of the overall processing time) and exactly one candidate activity of each pool has to be chosen. The binary linear program for each route takes the following mathematical form:

$$\begin{array}{ll} max & \vec{b}^T\vec{x} \\ s.t. & \begin{pmatrix} 1\cdots1 & & 0\cdots0 \\ \vdots & \ddots & \vdots \\ 0\cdots0 & & 1\cdots1 \\ & \vec{c}^T & \end{pmatrix}\vec{x} = \begin{pmatrix} 1 \\ \vdots \\ 1 \\ C_{max}-\Delta c \end{pmatrix} \\ & \vec{x} \in \{0,1\}^N \end{array} \qquad 3.5$$

This equation system describes a multiple choice knapsack. Moreover, $\vec{b}$ and $\vec{c}$ denote the column vector of benefits and costs. N represents the number of all candidates on a path. If each pool holds a constant amount of candidates c, then $N = cp$. In this case, a total enumeration yields c^p possibilities. Hence, some better strategy has to be found – a first consideration decides on the type of algorithm for finding a solution to this problem.

Regarding their output, optimisation procedures can be divided into three categories: namely in exact, approximative and heuristic ones. While the former specifies a method to locate the global optimum if it exists, an approximation algorithm provides provable quality and/or runtime bounds as opposed to an ordinary heuristic quickly detecting a good solution. The approximative heuristic presented in this work emerges from the programming relaxation of the linear integer program and has a proven accuracy with an empirical determined statistical dispersion of the mean execution time (chapter 5).

3.2 Simplex Algorithm

The method proposed in this section can be seen as a reasonable tradeoff between exactness and efficiency. As already discussed in chapter 2, the worst case time complexity for solving 3.5 can be pushed to $O(Nlog(N))$ by calculating the convex hull (Yang et al. [YTX+07]). However, no provable lower quality limit is indicated by the authors although the experiments have revealed good results for relatively small problems.

On the other hand, searching for the optimum theoretically causes exponential runtime. A practical approach depends heavily on the implementation and specific instance to solve. Branch-and-Bound using linear programming relaxations and cutting plane methods (both together subsumed under the term branch-and-cut) have turned out to be most promising as it has been the case for solving large Travelling Salesman Problems (Applegate et al. [ABCC98]).

Sharing linear programming relaxations as core concept between an approximative and an exact algorithm would be a good design criterion of the library from a software architectural point of view. Furthermore, this enhancement would allow switching of procedures e.g. if the optimal solution isn't found in a given time a rollback to the best relaxation as input for the heuristic would be possible. An exact method hasn't been implemented yet, important considerations are treated in section 3.3.

Neglecting the integrity constraint in 3.5, the optimum of this relaxation can be determined in polynomial time. Unfortunately, the simplex algorithm has been verified to be of exponential worst case complexity. Contrariwise, ellipsoid and interior point methods are bounded by a polynomial and the latter are even considered competitive with the simplex in most applications (Overton [Ove97]).

Concerning the implementation, the simplex algorithm has been chosen not only because of a possible extension to a branch-and-bound approach but also as a consequence of the problem's special structure. It will be shown in the following sections that each iteration's output is a vector with $p+1$ entries. Thus, on the termination every pool has to be represented by at least one candidate (exactly one pool eventually by two). So it becomes evident that not less than $p-1$ components will be integer at the end.

3.2.1 General Functionality

This subsection covers the basic principles of linear programming necessary for a transition and constructive discussion on the standard and revised simplex method – two concrete applications. A more comprehensive introduction can be found in Chvátal [Chv83] and Vanderbei [Van01].

First of all, the linear programming relaxation emerging from 3.5 is specified below. In this so called "augmented form" the artificial slack variable x_{N+1} is an additional element of $\vec{x}$ represented as an extra column in the coefficient matrix. Therefore it causes the costs (Δc) and an extra 0-entry in $\vec{b}$. The fact that every variable except x_{N+1} must not exceed 1 can be seen as ramification to the first p equations and the non-negativity constraint.

$$\begin{array}{ll} max & \vec{b}^T\vec{x} \\ s.t. & \begin{pmatrix} 1\cdots1 & & 0\cdots0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0\cdots0 & & 1\cdots1 & 0 \\ & \vec{c}^T & & 1 \end{pmatrix}\vec{x} = \begin{pmatrix} 1 \\ \vdots \\ 1 \\ C_{max} \end{pmatrix} \\ & \vec{x} \in (\mathbb{R}^+ \cup \{0\})^{N+1} \end{array} \qquad 3.6$$

The first task aims at the determination of a feasible start solution, provided that there exists one at all. For some programs, an auxiliary problem needs to be solved typically with the simplex itself (phase one of the two-phase simplex method). In this case, it reduces to storing those activities with minimal costs during the read-in. If the sum of minimal costs is already above a supposed value, the evaluation is aborted due to infeasibility. Other possibilities for a shortcut are outlined in chapter 4.

Once an initial solution has been determined for 3.6, the value of the objective function $\vec{b}^T\vec{x}$ is incremented within each iteration until the maximum has been reached or unboundedness has resulted. Geometrically, the solution space to the linear system of equations has the shape of a convex N-dimensional polyhedron. The optimum complies with the intersection of the hyperplane $\vec{b}^T\vec{x}$ (displaced as much as possible along its normal vector $\vec{b}$) and the polytope. For an arbitrary linear program this situation is shown in figure 2.

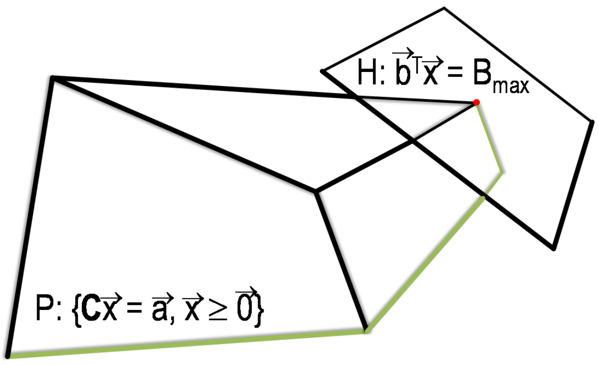

H: $\vec{b}^T\vec{x} = B_{max}$
P: $\{\mathbf{C}\vec{x} = \vec{a}, \vec{x} \geq \vec{0}\}$

Figure 2: The geometrical approach of solving $max\{\vec{b}^T\vec{x} \mid \boldsymbol{C}\vec{x} = \vec{a}, \vec{x} \geq \vec{0}\}$. The way the simplex might take from the start solution (leftmost) to the optimum (red) is marked green.

In case of no unboundedness, the intersecting set is either empty (infeasibility) or corresponds to a v-dimensional geometrical figure ($0 \leq v \leq N-1$). So if $v = 0$, only one single solution exists otherwise there are infinite many. Regarding the simplex algorithm it is important, that for every value of v at least one optimum has to reside at a vertex. The latter is defined as an intersection of all neighbouring facets, or in other words, $p+1$ variables are necessary for its representation being referred to as “basic”. If any of them happens to be 0, the basic solution is said to be degenerated. As a consequence, the objective value may not change during an iteration and the simplex may even start to cycle. A strategy to prevent cycling is presented at the end of this subsection.

The start solution determined as stated above, corresponds to a vertex. Within each iteration, a non-basic variable (that would increase the overall benefits) is selected for entering the basis (x_e), while a suitable basic one is made to leave it (x_l). To render this more precisely, the linear program relaxation in augmented form can be separated into basic (set B) and non-basic (set N) variables:

$$max\{\vec{b}^T\vec{x} \mid \boldsymbol{C}\vec{x} = \vec{a}, \vec{x} \geq \vec{0}\} \tag{3.7}$$

$$max\{\vec{b}_B^T\vec{x}_B + \vec{b}_N^T\vec{x}_N \mid \boldsymbol{C_B}\vec{x}_B + \boldsymbol{C_N}\vec{x}_N = \vec{a}, \vec{x} \geq \vec{0}\} \tag{3.8}$$

$$max\{B^* + \vec{b}_N^T\vec{x}_N \mid \boldsymbol{C_B}\vec{x}_B = \vec{a} - \boldsymbol{C_N}\vec{x}_N, \vec{x} \geq \vec{0}\} \tag{3.9}$$

$$max\{B^* + \vec{b}_N^T\vec{x}_N \mid \vec{x}_B = \boldsymbol{C_B}^{-1}\vec{a} - \boldsymbol{C_B}^{-1}\boldsymbol{C_N}\vec{x}_N, \vec{x} \geq \vec{0}\} \tag{3.10}$$

$$max\{B^* + \vec{b}_N^T\vec{x}_N \mid \vec{x}_B = \vec{x}_B^* - \boldsymbol{C_B}^{-1}\boldsymbol{C_N}\vec{x}_N, \vec{x}_B \geq \vec{0}, \vec{x}_N = \vec{0}\} \tag{3.11}$$

Especially the last equation can be interpreted as a snapshot of a current iteration (confirmed by B^* and $\vec{x}_B^*$ indicating the actual sum of benefits and the intermediate values of $\vec{x}_B$).

Possible candidates for x_e are formally characterised by $\{x_i \mid b_i > 0, i \in N\}$. If this set happens to be empty, then the optimum has been detected and the execution stops. Otherwise, x_e takes the highest value permitted, while a basic variable gets 0. To decide which one, the expression for $\vec{x}_B$ in 3.11 should be considered:

$$\vec{x}_B^* - \boldsymbol{C_B}^{-1}\boldsymbol{C_N}\vec{x}_N \geq \vec{0} \tag{3.12}$$

$$\vec{x}_B^* - x_e\vec{d} \geq \vec{0} \qquad \vec{d} := \left(\boldsymbol{C_B}^{-1}\boldsymbol{C_N}\right)_e = \boldsymbol{C_B}^{-1}\vec{c}_e \tag{3.13}$$

$$x_e = min\{\frac{x_{Bi}^*}{d_i}, i \in B\} \tag{3.14}$$

Formula 3.13 uses the fact that $\vec{x}_N$ contains the value of x_e as the only hopefully positive entry. Hence, $\vec{d}$ is defined by the vector of the entering column $\vec{c}_e = \left(\boldsymbol{C_N}\right)_e$. Due to $\vec{x}_B \geq \vec{0}$, maximising x_e means setting it to the minimum quotient. The subscript that fulfils the latter therefore determines the leaving variable x_l as a candidate of $\{x_i \mid x_e = min\frac{x_{Bi}^*}{d_i}, i \in B\}$.

As opposed to the set for x_e, the one for x_l cannot be empty. However, two kinds of anomalies may result from the term $min_{i \in B} \frac{x^*_{Bi}}{d_i}$:

$$x_e = \begin{cases} 0 & \text{degeneracy} \\ \infty & \text{unboundedness} \end{cases} \qquad 3.15$$

If x_e can be chosen arbitrarily large (and so also the objective value) then the linear program at hand is unbounded and the simplex algorithm terminates signalling an error state. Degeneracy poses a bigger problem: It slows down execution if the benefits aren't increased and may even lead to the occurrence of cycles. To prevent cycling, Bland's smallest-subscript rule as described and proved in Chvátal [Chv83] can be applied:

"The simplex method terminates as long as the entering and leaving variables are selected by the smallest-subscript rule in each iteration."

This yields the following formulas for unambiguously selecting x_e and x_l out of the above identified sets:

$$x_e: \quad e = min\{i \mid b_i > 0, i \in N\} \qquad 3.16$$

$$x_l: \quad l = min\{i \mid x_e = min \frac{x^*_{Bi}}{d_i}, i \in B\} \qquad 3.17$$

In the following subsection pivoting, the mathematical equivalent for traversing along an edge, is visualised by using a tableau. This procedure, known as standard simplex method, reflects the geometrical concept of an iteration in a straight-forward way (Sedgewick [Sed83]). The inherent disadvantages of an implementation for large instances are outlined thereafter.

3.2.2 Standard Simplex Method

The pivot operation consists of finding a certain pivot element and a subsequent update process. The former is the intersection of the pivot column (entering variable) and the pivot row (leaving variable). Similar to pivoting in Gaussian elimination, the pivot row is divided by the pivot number and an appropriate multiple is added to every other row, so that the pivot column equals the unit vector of the entering variable afterwards.

To clarify things, the first iteration of the standard simplex method (moving from the start vertex to a neighbouring) is presented below with concrete numbers for $max\{\vec{b}^T\vec{x} \mid \boldsymbol{C}\vec{x} = \vec{a}, \vec{x} \geq \vec{0}\}$. The example contains the case that a basic solution gets degenerated and the final output has to be integer (i.e. x_6 and x_{12}, the candidates with maximum benefits in the pools).

$$\begin{aligned}
\vec{a} &= \begin{pmatrix} 1 & 1 & 100 \end{pmatrix}^T \\
\vec{b}^T &= \begin{pmatrix} 133 & 135 & 133 & 142 & 142 & 161 & 133 & 146 & 125 & 144 & 136 & 197 & 0 \end{pmatrix} \\
\boldsymbol{C} &= \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 50 & 92 & 42 & 66 & 38 & 96 & 62 & 5 & 86 & 47 & 14 & 1 \end{pmatrix}
\end{aligned} \qquad 3.18$$

At the beginning $\vec{x}_B^* = \begin{pmatrix} 1 & 1 & 95 \end{pmatrix}^T$ because x_1, x_9 and x_{13} (the slack variable) are in the basis. The overall benefits B^* amount to 258. Several representations of the tableau are possible (e.g. as in Sedgewick [Sed83]), a modified one for an arbitrary start solution looks as follows:

$$\begin{array}{r|rrrrrrrrrrrrr}
-258 & 0 & 2 & 0 & 9 & 9 & 28 & 8 & 21 & 0 & 19 & 11 & 72 & 0 \\
\hline
1 & 1 & \boxed{1} & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
95 & 0 & 50 & 92 & 42 & 66 & 38 & 91 & 57 & 0 & 81 & 42 & 9 & 1
\end{array} \qquad 3.19$$

The first row in 3.19 results from subtracting 133 from all benefit values in the first pool, 125 from each of the other and the sum (258) from 0. This does not affect the slack variable which is therefore left at 0. The second and third row comply with the coefficients in the equation system. In the last one, the candidates' costs have been diminished by those of the selected ones and the difference to C_{max} computes to $100 - 5 = 95$. All basic variables correspond to unit vectors in the tableau. The pivot element in the box initiates the first update.

The single steps of an iteration are outlined below. The first two represent the selection of a vertex to head for while the last one defines the transition along this specified edge:

1. choosing the pivot column (entering variable e.g. according to 3.16)
2. selecting the pivot row (leaving variable e.g. compliant with 3.17)
3. transforming the column vector of the new basic variable into a unit vector (same operation as in Gaussian elimination)

Applying Bland's rule, the determination of the pivot number t_{le} yields 1 for $e = 2$ and $l = 1$. So row 1 remains as before, its multiples -2 and -50 have to be added to the first and last row respectively. The second tableau takes the following form. The example is completed in appendix A.

$$\begin{array}{r|rrrrrrrrrrrrr}
-260 & -2 & 0 & -2 & 7 & 7 & 26 & 8 & 21 & 0 & 19 & 11 & 72 & 0 \\
\hline
1 & 1 & 1 & 1 & \boxed{1} & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
45 & -50 & 0 & 42 & -8 & 16 & -12 & 91 & 57 & 0 & 81 & 42 & 9 & 1
\end{array} \tag{3.20}$$

At first sight, it might seem sufficient to store and update only the first and last row. However, things change when the slack variable leaves the basis. Thus, memory requirements increase and runtime declines with the tableau's size. Contrariwise, large programs deserve efficient computer implementations using as little storage as possible and additionally providing numerical stability when the entries get fractional.

3.2.3 Revised Simplex Method

The revised simplex method addresses these demands by uncoupling the simplex algorithm from a tableau. From the latter, only the basic variables are used. An extensive description of this application can be found in Chvátal [Chv83].

The separation into basic and non-basic variables leads to equation 3.11. Substituting $\boldsymbol{C_B}^{-1}\vec{a} - \boldsymbol{C_B}^{-1}\boldsymbol{C_N}\vec{x}_N$ for $\vec{x}_B$ shows the linear programming relaxation (augmented form) in a slightly different way:

$$\begin{aligned} max \quad & \vec{b}_B^T \boldsymbol{C_B}^{-1}\vec{a} + \left(\vec{b}_N^T - \vec{b}_B^T \boldsymbol{C_B}^{-1}\boldsymbol{C_N}\right)\vec{x}_N \\ s.t. \quad & \vec{x}_B = \boldsymbol{C_B}^{-1}\vec{a} - \boldsymbol{C_B}^{-1}\boldsymbol{C_N}\vec{x}_N \\ & \vec{x}_B \geq \vec{0}, \vec{x}_N = \vec{0} \end{aligned} \qquad 3.21$$

Compared to 3.11, $\vec{b}_B^T \boldsymbol{C_B}^{-1}\vec{a} = B^*$ and $\boldsymbol{C_B}^{-1}\vec{a} = \vec{x}_B^*$. Moreover, $\vec{d}$ has been introduced in equation 3.13 as $\boldsymbol{C_B}^{-1}\vec{c}_e$ with $\vec{c}_e$ denoting the vector of the entering column. Defining $\vec{y}^T$ as $\vec{b}_B^T \boldsymbol{C_B}^{-1}$ in addition, the new formulation of the problem is:

$$\begin{aligned} max \quad & B^* + \left(\vec{b}_N^T - \vec{y}^T \boldsymbol{C_N}\right)\vec{x}_N \\ s.t. \quad & \vec{x}_B = \vec{x}_B^* - x_e\vec{d} \\ & \vec{x}_B \geq \vec{0}, \vec{x}_N = \vec{0} \end{aligned} \qquad 3.22$$

With the same geometric interpretation as above, an iteration is determined by identifying a variable to enter the basis (which in turn decides on another one to leave it) and its subsequent update:

1. calculating $\vec{y}^T$ in $\vec{y}^T \boldsymbol{C_B} = \vec{b}_B^T$ to get the first component of the vector $\vec{b}_N^T - \vec{y}^T \boldsymbol{C_N}$ greater than 0 (entering variable with Bland's rule)
2. solving $\boldsymbol{C_B}\vec{d} = \vec{c}_e$ for $\vec{d}$ needed in 3.17 (leaving variable)
3. replacing $\vec{x}_B$ with $\vec{x}_B^* - x_e\vec{d}$ and the newly emerging 0-entry with x_e

In general, solving the two equation systems $\vec{y}^T \boldsymbol{C_B} = \vec{b}_B^T$ and $\boldsymbol{C_B}\vec{d} = \vec{c}_e$ from scratch compares to a computational bottleneck. Chvátal [Chv83] explains the eta factorisation of $\boldsymbol{C_B}$ with different devices – the simplest among them is a variation of the well-known "product form of the inverse". Since the basis matrix in iteration k differs from a preceding in just one column (m), the plain relationship between them looks as follows:

$$\boldsymbol{C_{B_k}} = \boldsymbol{C_{B_{k-1}}}\boldsymbol{E_k} \qquad 3.23$$

$\boldsymbol{E_k}$ constitutes the identity matrix whose m^{th} column has been replaced by $\vec{d}$. As $\boldsymbol{C_{B_k}} = \boldsymbol{C_{B_0}}\boldsymbol{E_1}\cdots\boldsymbol{E_k}$, $\vec{y}^T \boldsymbol{C_{B_k}} = \vec{b}_B^T$ and $\boldsymbol{C_{B_k}}\vec{d} = \vec{c}_e$ can be evaluated iteratively with each bracket term taking only little time:

$$\left(\left(\left(\vec{y}^T \boldsymbol{C_{B_0}}\right) \boldsymbol{E_1}\right) \cdots\right) \boldsymbol{E_k} = \vec{b}_B^T \qquad 3.24$$

$$\boldsymbol{C_{B_0}} \left(\boldsymbol{E_1} \left(\cdots \left(\boldsymbol{E_k} \vec{d}\right)\right)\right) = \vec{c}_e \qquad 3.25$$

Further refinements and the occasional refactorisation process of the basis to shorten the number of eta matrices are described in Chvátal [Chv83]. Fortunately, $\vec{y}^T$ and $\vec{d}$ can always be determined directly for 3.6 because each $\boldsymbol{C_B}$ represents the identity matrix except for at most two rows. Hence, no more than two unknowns have to be computed for these vectors. Implementation details are revealed in chapter 4, the first iteration of the revised simplex method for the same example 3.18 is demonstrated at once. It is finished in appendix A.

$$\vec{x}_B^* = \begin{pmatrix} 1 \\ 1 \\ 95 \end{pmatrix}, \qquad \boldsymbol{C_{B_0}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 5 & 1 \end{pmatrix} \qquad 3.26$$

<u>Iteration 1:</u>

1. $$\vec{y}^T \boldsymbol{C_{B_0}} = \vec{b}_{B_0}^T$$
$$\vec{y}^T = \begin{pmatrix} 133 & 125 & 0 \end{pmatrix}$$
$$b_2 - \vec{y}^T \vec{c}_2 = 2, \qquad \Rightarrow x_2 \text{ enters the basis}$$

2. $$\boldsymbol{C_{B_0}} \vec{d} = \vec{c}_2$$
$$\vec{d} = \begin{pmatrix} 1 & 0 & 50 \end{pmatrix}^T \qquad 3.27$$
$$\vec{x}_B^* - x_e \vec{d} \geq \vec{0}$$
$$x_e = 1, \qquad \Rightarrow x_1 \text{ leaves the basis}$$

3. $$\vec{x}_B^* = \begin{pmatrix} 1 \\ 1 \\ 45 \end{pmatrix}, \qquad \boldsymbol{C_{B_1}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 50 & 5 & 1 \end{pmatrix}$$

3.2.4 Numeric Stability

Numeric accuracy plays an important role in floating-point arithmetic. Rounding errors which are inherent to the use of data types like “float” or “double” and their propagation have to be prevented as far as possible. An overview of this topic is provided in Goldberg [Gol91].

In the context of linear equation systems, Chvátal [Chv83] points out that rounding errors may not only accumulate in long chain calculations to useless results but also for the instance given below:

$$\begin{aligned} 0.0001z_1 &+ & z_2 &= 1 \\ 0.5z_1 &+ & 0.5z_2 &= 1 \end{aligned} \qquad 3.28$$

Pivoting (i.e. dividing the first row by 0.0001 and adding a suitable multiple of it to the second one such that z_1 is eliminated therein) and rounding to three decimal places leads to $z_2 = 1$ and $z_1 = 0$, although $z_1 = z_2 = 1$ would be correct.

The reason for this behaviour lies in the small pivot element (relative to the other coefficients). Therefore ε_2, a positive number close to 0, excludes a division if the divisor is below that limit. A second one used in the library, ε_1, prevents a variable from entering the basis if it happens to be indistinguishable from 0. Murtagh [Mur81] specifies $\varepsilon_1 = 10^{-5}$ and $\varepsilon_2 = 10^{-8}$ suitable with the precision of the “double” data type.

To control error propagation, the “double” values of the basis matrix need to be refactored periodically. The length of the period depends on the nature of the data and may start as soon as the slack variable is out of the basis, because two candidates appear in $\vec{x}_B^*$ for one pool then. The adjustment essentially consists of comparing its actual and expected costs. The formula together with the code of method “refactor” are part of section 4.1 in the next chapter.

Rounding errors could be completely avoided by computing exclusively with integer data types. This is arranged e.g. by storing nominator and denominator for a fraction number separately or by transforming to an “int” or “long” (multiplication with the reciprocal for the suitable accuracy). However, these techniques cause an extra overhead of memory and runtime. Thus their preference to floating point operations (which on the other hand take longer than their corresponding counterparts) remains to be justified.

3.3 Exact Methods

In the last section of this chapter, an approach to the exact solution finding process is sketched. The latter would be an architecturally facilitated enhancement of the library due to the usage of linear programming relaxations in the enumeration tree. Exact methods may not only serve for bridging the accuracy gap encountered for a small number of nodes (5.1), but also provide the user with the best configuration on demand. If a time constraint (e.g. realised as a predefined limit of calls to method “simplex”) is exceeded, a fallback to the actually best integer solution remains.

Several approaches exist for the realisation, among all of them, the branch-and-bound method with a good branching scheme as explained in Nemhauser et al. [NW88] seems to be most useful. The generalised branching is illustrated in figure 3. Splitting the index set P_i in two halves P_{i1} and P_{i2} with (almost) equal cardinality guarantees more progress than singling out a variable. Furthermore, it can be verified easily that the number of paths complies with all possible valid solutions.

The rounded relaxation determined by the heuristic constitutes a first lower limit. Before starting the enumeration it can be improved in different ways. First, the candidate that has been chosen in the pool with the two fractional variables can probably be replaced by a better suited one. Second, a cost limit can be figured out logarithmically for which the approximation algorithm yields an improved lower bound. For that purpose, the costs can be narrowed to $[c_{min}, 2c_{min}]$ resulting into $ld(c_{min})$ simplex-calls at most.

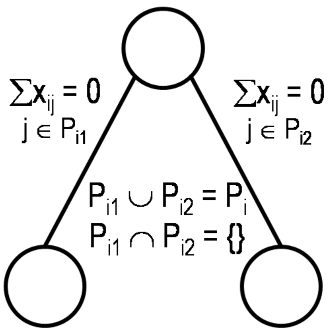


Figure 3: A generalised branching rule for enumeration. The subsets of P_i are preferably of (almost) equal size.

These techniques help to keep the branch-and-bound tree statically short. During the enumeration process, cutting-planes may contribute to the speed. But only those should be considered that tighten the bounds of a single variable's codomain to preserve the structure of the coefficient matrix. A good combination of these aspects shall be part of a future release.

4 Implementation

The implementation of the algorithm and the processing of an abstract workflow in common, together with important means of exporting and testing the library's functionality shall be the topic of this chapter. First of all, some general information is given in advance. The three parts of the software are analysed afterwards. Finally, necessary instructions for its usage are presented.

WorkflowCompositionLibrary is written in Visual C, an adaptation to the ANSI C standard is planned in a forthcoming release. The motivation of preferring C over a popular object-oriented programming language like C# or Java is mainly justified by the flexible memory management. It permits an adjustable reservation of storage and frees it instantly on demand (no garbage collection). Secondly, pointer arithmetic (i.e. the advantage of modifying pointers instead of moving data) identifies speed issues as a third reason for choosing C.

WorkflowCompositionWrapper implemented in C#, controls the access to the library's "export" functions instead of invoking the specified methods directly. Its main purpose lies in wrapping the unmanaged code in WorkflowCompositionLibrary which cannot be referenced as resource by projects containing managed code (Keserovic et al. [KMN03]). Furthermore, it adds object-oriented behaviour by encapsulating the functionality into an operator object whose single instance permits its utilisation (exception handling inclusive) to one application at the same time. Thus it enables an easy use of the library and keeps the caller's code clean.

WorkflowCompositionTest contains NUnit test methods to be performed on the operator. Their task is to correctly process small, medium, large and invalid instances. A high test coverage is achieved not only with selected predefined tests but also by generating randomly routes and whole graphs. The latter allow close runtime studies in addition. Moreover, the test cases together with a simulated call from another application storing its data in a simplified activity graph, demonstrate how to work with the library through an operator instance.

4.1 WorkflowCompositionLibrary

WorkflowCompositionLibrary comprises the read-in procedure and the evaluation part. The former describes how to initialise and refresh variables and fields before any activity is actually inserted. The latter can be one of three types, namely single, pool or candidate which correspond to “insertsa”, “insertpa” and “insertca” respectively. After this route has been completely read into the library, a certain amount of “back statements” (“insertbs”) enables the insertion of a new activity at the right place on the new path. Before that happens, the algorithm is run on the old one. Once the processing of the last route has been initiated by a call to “result”, the resources may be freed with “clean” or the variables can be reinitialised for a subsequent use (eventually for an update).

4.1.1 Read-in

The functions for export, i.e. the ones being publicly available and therefore identified as “extern”, are declared in “export.h”. Their complete method signatures are shown in listing 1.

```
extern DLLEXPORT void init(const int _num_activities, const int _num_pools, const int
    _num_candidates, const int _max_cost, const int _min_benefit, void (*_exception)(
    const char *message), const unsigned char _print);

extern DLLEXPORT void refresh(const int _max_cost, const int _min_benefit, const
    unsigned char _print);

extern DLLEXPORT void insertsa(const int id, int benefit, int cost);

extern DLLEXPORT void insertpa();

extern DLLEXPORT void insertca(const int id, int benefit, int cost);

extern DLLEXPORT void insertbs(const int steps);

extern DLLEXPORT int *result();

extern DLLEXPORT void clean();
```

Listing 1: Declaration of export functions

The initialisation asks for a specification of the number of single activities (S), num_pools (P) and num_candidates (C). The former is implicitly available through $S = A - CP$ where A stands for the total number of activities on a path. If any of these user-defined values turns out to be in an inappropriate range, it is adjusted accordingly as can be seen in listing 2. The constants MAX_ACTIVITIES, MAX_POOLS and MAX_CANDIDATES are predefined in the common header file (appendix B). Their selected magnitudes make the memory resources being covered by 256MB RAM. Adapted use of storage through "malloc" in "init" is explained in section 5.2.

```
num_activities = (_num_activities > 0 && _num_activities < MAX_ACTIVITIES) ?
    _num_activities : MAX_ACTIVITIES;
num_candidates = (_num_candidates > 0 && _num_candidates < MAX_CANDIDATES) ?
    _num_candidates : MAX_CANDIDATES;
num_pools = num_activities / num_candidates;
if (_num_pools > 0 && _num_pools < MAX_POOLS)
{
    if (num_pools > _num_pools)
    {
        num_pools = _num_pools;
    }
}
else
{
    if (num_pools > MAX_POOLS)
    {
        num_pools = MAX_POOLS;
    }
}
```

Listing 2: Setting the number of activity types

At the end of "init", method "refresh" is implicitly invoked (which has to be done explicitly by the user later on). The parameters _max_cost, _min_benefit and _print specifying C_{max}, b_{min} and the print option, are passed to this method as arguments. The function pointer void (*exception)(const char *message), also initialised in "init" is declared in "export.h" together with function "leave". In the latter, it causes an exception in the wrapper where it constitutes a delegate.

Having set the values for internal variables in "refresh", three different types of activities can be inserted. For singles and candidates, this means to store their parameter values in an array element of type activity which is a 12 byte structure. Trying to read in a candidate without a pool, results into an error – the 16 byte pool needs to be assigned to an entry of pools. The struct pool and struct activity are defined in “common.h” and listed together with the code of “insertsa”, “insertpa” and “insertca” in appendix B.

The output can be retrieved by calling “result”. It is assumed that the last route has been read into the library. The execution of “insertpa” steers the processing of the last path by setting “deltah”, the height difference to one less than the number of nodes. Specifying the steps to go back for starting another branch at the designated position is usually done with “insertbs”. The condition deltah != 0 is checked in “insertsa” and “insertpa”. If this expression evaluates to “true”, the optimisation procedure is launched, where necessary and printing gets started before the array indices (not the elements themselves) are resetted (in “reset”). Finally, “clean” frees all memory allocated in “init”.

4.1.2 Algorithm

Function “trigger” represents the entry point to the algorithm. Inside, “lbenefit” and “lcost”, the lower bound of benefits and costs are calculated. An invocation of “shortcut” verifies, if a solution can be obtained without the use of “simplex”. In case that a new global optimum has been found, “lbenefit” and “lcost” are updated and the ids are stored into the “rids” array in “fillr”. The signatures of the private methods of “algorithm.c” are shown below (listing 3) and explained subsequently. The source code of “trigger” is given in appendix B.

```
int shortcut();
void fillr();
int simplex();
void step0();
int step234(int *_k, int *_l, int *_j, int _c1, int _c2, double *_t, const double yL);
void refactor();
int eval();
void opt();
```

Listing 3: Members of algorithm.c

The purpose of “shortcut” lies in quickly examining the problem if it’s

1. infeasible: no solution exists
2. already below a predetermined bound: a previously completed route's outcome remains preeminent
3. easily derivable: consistency with the selection of all candidates having maximal benefits or minimal costs

The implementation of the revised simplex method forms the central part. The return type of "simplex" is "int" because the linear programming relaxation may be unbounded theoretically. At the beginning, "step0" is executed for initialising "basis" (a two-dimensional array storing pairs of pool index and offset), "sB" (an array to get the correct index of "basis" for a certain index of a non-empty pool) and "xB" (the array with the values of the basis variables) with the start solution of minimal costs (appendix B).

The algorithm comprises three parts as outlined in 3.2.3. In the code, they have been flattened to five steps, each corresponding to an essential operation to be performed. First, $\vec{y}^T$ needs to be figured out from $\vec{y}^T \boldsymbol{C_B} = \vec{b}_B^T$. Depending whether the slack variable is in the basis or not, $\boldsymbol{C_B}$ consists either of p or $p-1$ row unit vectors. In the first case, the equation system for $\vec{y}^T$ can be solved directly while in the other, two equations with two unknowns have to be computed in advance (listing 4). The formula for the entry $y[j]$ and $y[bidx]$, with j being the index of the pool (represented with two candidates in $\boldsymbol{C_B}$) and $bidx$ indicating the last row, can be deduced as follows:

$$y[j] + c_1 y[bidx] = b_1 \qquad 4.1$$

$$y[j] + c_2 y[bidx] = b_2 \qquad 4.2$$

$$y[bidx] = \frac{b_1 - b_2}{c_1 - c_2} \qquad 4.3$$

$$y[j] = \frac{c_1 b_2 - b_1 c_2}{c_1 - c_2} \qquad 4.4$$

Here, b_1, c_1 and b_2, c_2 mark the benefits and costs of the two candidates belonging to the same pool. A necessary condition for the use of these formulas is $c_1 \neq c_2$. If $c_1 = c_2$, then $\boldsymbol{C_B}$ would be singular. The impossibility of this situation is proved in Chvátal [Chv83]. Once 4.3 and 4.4 have been applied, the other entries $y[k]$ are given by $b_k - y[bidx]c_k$ where b_k and c_k constitute the benefit and cost value of the k^{th} pool's candidate in the basis.

```c
/* pool j appears twice among basis variables */

/* basis[0][idx1] == basis[0][idx2] */
j = basis[0][idx1];

b1 = (activities + (pools + j)->idx + basis[1][idx1])->benefit;
b2 = (activities + (pools + j)->idx + basis[1][idx2])->benefit;
c1 = (activities + (pools + j)->idx + basis[1][idx1])->cost;
c2 = (activities + (pools + j)->idx + basis[1][idx2])->cost;

/* c1 != c2, otherwise the basis matrix would be singular which is not possible
     (proof can be found in Chvátal: Linear Programming) */
yL = ((double)(b2 - b1)) / (c2 - c1);
y[bidx] = yL;
y[j] = ((double)(b1 * c2 - c1 * b2)) / (c2 - c1);

for (i = 0; i <= bidx; i++)
{
    k = basis[0][i];
    if (k != j)
    {
        y[k] = (activities + (pools + k)->idx + basis[1][i])->benefit -
            yL * (activities + (pools + k)->idx + basis[1][i])->cost;
    }
}
```

Listing 4: Solution to $\vec{y}^T \boldsymbol{C_B} = \vec{b}_B^T$ (slack variable has left the basis)

Step 2 in "step234" makes a decision on a suitable variable for entering the basis. The current version implements Bland's rule (introduced in 3.2.1) which avoids cycling. However, the number of iterations could grow very large, especially if the candidates are sorted ascending by b/c. Therefore, a better pricing strategy would lead to a considerable reduction of runtime. A good suggestion that does not involve intensive computations (as opposed to "steepest edge") would be a descendent ordering by b/c as preprocessing for the simplex.

The determination of $\vec{d}$ in $\boldsymbol{C_B}\vec{d} = \vec{c}_e$ takes place in step 3. If the slack variable resides in the basis then $d[sB[k]]$ is set to either 1 (k equals the actual pool index) or 0, when iterating over k. The expression $sB[k]$ evaluates to the right column index for the non-empty pool k. The entry at position $idx1$ (= $idx2$) can be fixed at $c_e - c_l$, the difference between the entering and leaving variable's costs. Similar to $\vec{y}^T$, $\vec{d}$ requires to solve two equations if $\boldsymbol{C_B}$ has only $p-1$ row unit vectors (listing 5). The corresponding $p-1$ elements are 1 or 0 as above and substituted in the last row. However, three cases have to be differentiated:

1. the entering variable belongs to the same pool as the fractionals
2. the slack variable is going to enter the basis
3. the entering variable is a member of a pool with just one candidate in the basis

```
/* one pool appears twice among basis variables */

for (i = 0; i <= pidx; i++)
{
    if (sB[i] != -1 && i != basis[0][idx1])
    {
        d[sB[i]] = (i == l) ? 1 : 0;
    }
}

/* c1 != c2, otherwise the basis matrix would be singular which is not possible
    (proof can be found in Chvátal: Linear Programming) */
if (basis[0][idx1] == l)
{
    d[idx1] = ((double)(activities + p->idx + j)->cost - c2) / (c1 - c2);
    d[idx2] = (c1 - (double)(activities + p->idx + j)->cost) / (c1 - c2);
}
else if (l == pidx + 1)
{
    d[idx1] = 1.0 / (c1 - c2);
    d[idx2] = 1.0 / (c2 - c1);
}
else
{
    c = (activities + p->idx + basis[1][sB[l]])->cost;
    d[idx1] = ((double)(activities + p->idx + j)->cost - c) / (c1 - c2);
    d[idx2] = (c - (double)(activities + p->idx + j)->cost) / (c1 - c2);
}
```

Listing 5: Solution to $\boldsymbol{C_B}\vec{d} = \vec{c}_e$ (slack variable is not in the basis)

The formulas for the remaining entries $d[idx1]$ and $d[idx2]$ (with $idx1$ and $idx2$ being the indices of the two fractional variables) are declared below. They can be verified with the example in appendix A (e.g. iteration 5, 8 and 9).

$$1. \quad d[idx1] = \frac{c_e - c_2}{c_1 - c_2} \quad d[idx2] = \frac{c_1 - c_e}{c_1 - c_2} \tag{4.5}$$

$$2. \quad d[idx1] = \frac{1}{c_1 - c_2} \quad d[idx2] = -d[idx1] \tag{4.6}$$

$$3. \quad d[idx1] = \frac{c_e - c_l}{c_1 - c_2} \quad d[idx2] = -d[idx1] \tag{4.7}$$

Step 4 deals with searching for the leaving and to settle a value for the entering variable as a consequence. If none can be found, the problem is unbounded which has not been experienced in the test runs though. Finally, “basis”, “sB” and “xB” are updated. The first two merely represent programming facilities, the new entries of “xB” follow from the algorithm of the revised simplex method (step 5).

To prevent the propagation of rounding errors as discussed in 3.2.4, the content of “xB” needs to be adjusted periodically. This is accomplished by invoking function “refactor” whenever the iteration counter equals a multiple of REF_PERIOD, a constant defined in the common header file. The refactorisation is organised in two parts: First, all elements except the possible fractionals are rounded to the nearest integer and their costs are added up. If the slack variable is not in the basis, the correct summand φ to actualise the fractional variables with, is calculated thereafter (listing 6). Supposing that $xB[idx1]'$ and $xB[idx2]'$ denote their actual values, then their pool causes the costs $c'_P = c_1 xB[idx1]' + c_2 xB[idx2]'$ contrary to the true data $xB[idx1]$ and $xB[idx2]$ with $c_P = c_1 xB[idx1] + c_2 xB[idx2]$:

$$c'_P - c_P = c_1(xB[idx1]' - xB[idx1]) + c_2(xB[idx2]' - xB[idx2]) \tag{4.8}$$

$$c'_P - c_P = c_1\varphi - c_2\varphi \tag{4.9}$$

$$\varphi = \frac{c'_P - c_P}{c_1 - c_2} \tag{4.10}$$

Equation 4.9 results from the fact that both variables always add up to 1. The entries $xB[idx1]$ and $xB[idx2]$ are therefore set to $xB[idx1]' - \varphi$ and $xB[idx2]' + \varphi$ respectively.

```
/* refactor fractional pool */

cP = acostl - cP;   /* cP is the correct cost value of the fractional pool */
c1 = (activities + (pools + basis[0][idx1])->idx + basis[1][idx1])->cost;
c2 = (activities + (pools + basis[0][idx2])->idx + basis[1][idx2])->cost;
f = (xB[idx1] * c1 + xB[idx2] * c2 - cP) / (c1 - c2);
xB[idx1] -= f;
xB[idx2] += f;
```

Listing 6: Refactorisation of the fractional variables

The main purpose of “eval” lies in the computation of the lower benefit and cost limit (“lbenefit” and “lcost”). As the solution tends to be non-integer in most cases, it has to be rounded as indicated below. Furthermore, the ids of the selected candidates are stored. This procedure is either part of “eval” or outsourced to “opt” for an already optimal linear programming relaxation.

$$xB[idx1] = \begin{cases} 0 & c_1 < c_2 \\ 1 & else \end{cases} \qquad 4.11$$

4.2 WorkflowCompositionWrapper

An instance of “Operator” embodies the wrapper’s functionality at runtime. To facilitate its usage for other applications, the code is discussed in detail and printed in appendix B.2.

An operator can be retrieved from the factory method “CreateInstance” with the expected number of total activities, pools and candidates as arguments. Moreover, C_{max}, b_{min} and if printing should be turned on, need to be specified. The Singleton Pattern assures only one active instance at a time to handle an application’s request. This prevents a modification of the data by another caller. The private constructor in turn invokes “init” with the user’s input and a previously added callback function (for raising an appropriate exception) to the delegate.

The principle of the wrapper methods is first to assign a correct exception type if any of it can be thrown at all, secondly to access the library and at least to reset “exceptionType” to 0. “ReturnResult” shows the “IntPtr” object as counterpart to the int-pointer in C. The latter is needed for “Marshal.ReadInt32” to get the output array’s elements iteratively. Its length is stored in the first position, the second one contains the non-empty solution’s overall benefit value. Therefore, “intArr” is either null (if no activities have been inserted) or it holds the latter at index 0 and the ids in sequence.

4.3 WorkflowCompositionTest

Once the preconditions in 4.4 are met, the test project launches the graphical user interface of NUnit on start. The “Main” method can be found in “Program.cs” and is listed below. The test classes have been added to a test suite in “AllTests.cs” which comprises 91 single test cases altogether. So, executing all of them will take some time depending on the current settings.

```
[STAThread]
static void Main(string[] args)
{
    NUnit.Gui.AppEntry.Main(new string[] { System.Reflection.Assembly.
        GetExecutingAssembly().Location });
}
```

Listing 7: Entry point of test project

Small, medium, large and invalid test instances (abbreviated “si”, “mi”, “li” and “ii”) have been designed to verify the program’s functionality. The approach consists of inserting activities first and to invoke the “ReturnResult” function of the operator thereafter. The latter is performed in the corresponding “ReturnResult” test classes. As the methods in “InsertActivityIITest.cs” provoke an exception, no “ReturnResultIITest.cs” for analysing the final outcome exists. The following assumptions of erroneous user inputs have been made:

- Insertion of activities at wrong positions
- Entering a candidate without a pool

- Read-in of too many activities, pools, candidates

A test is passed, if an exception of a specific type (thrown by the operator) occurs. As an example, listing 8 demonstrates a check on the program's correct behaviour in case of exceeding the maximum number of activities by one:

```
[Test]
[ExpectedException(typeof(InsertActivityException))]
public void Test04()
{
    int i;
    op.RefreshOperator(0, ACTIVITIES / 2, false);

    for (i = 0; i <= ACTIVITIES; i++)
    {
        op.InsertSingleActivity(i, i + 1, i);
    }
}
```

Listing 8: Overflow of activities

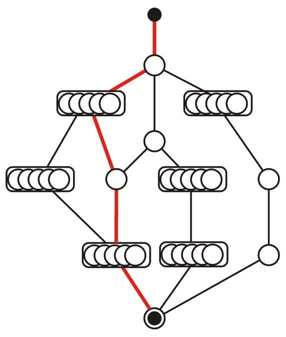
The common organisation of the test classes becomes evident when regarding the one for insertion first. All of them start with "Init" where an operator is retrieved. This instance remains usable until it is released and set to "null" in function "Clean". Within the single test cases, "RefreshOperator" has to be called before any activities will be read in. If they fail, an exception must have happened somewhere in the code and the error message is specified as argument to "AssertFail". Inherent to these classes is also a getter-method for the operator. This allows the overlaying tests for "ReturnResult" to access the same instance already used for the buildup of the workflow. A reference to the appropriate "InsertActivity" test class is obtained via the constructor at the beginning. After that, the test fixture setup reduces to the invocation of "Init" by the reference. Within a test method, the associated one is called first to query for the output thereafter. The test fixture tear down works analogue to "Init" in the "ReturnResult" test classes. Two example test files ("InsertActivitySITest.cs" and "ReturnResultSITest.cs") listed in appendix B.3 represent this relationship.

The difference between the tests for small, medium and large instances lies in their complexity. The former e.g. check simple scenarios like the insertion of:

- no activity
- a single activity
- a pool activity with 0, 1, 5 candidate(s)
- a single/pool activity with 2 children

The last test functions cover branching, i.e. the read-in of a graph consisting of more than one path and some special cases like all activities are equal or $b_{min} = 0$. Therefore, it is possible to control each element of the result array in “ReturnResultSITest.cs”. As opposed to these static tests, dynamically constructing graphs renders a verification of each entry in the outcome inconvenient (automatic tests) if not impossible (random tests), because the heuristic only finds an approximate solution. Instead, the relative error is confirmed to be less than the reciprocal number of nodes, a direct consequence of choosing the benefit value from the interval $[b_{min}, 2b_{min}]$ (section 5.1).

“InsertActivityMITest.cs” simulates how the operator is used efficiently in an overlaying application where the data has been stored into “ActivityGraph”. Similar to the representation in the library, an instance of “Activity” is either single, pool or candidate. Additionally, a list for the parents and one for children can be specified. A flag is set if a pool is constructed. A reference to the actual activity is assigned the field “candidate” of a previous one in case that a candidate has been read in. To sum up, “ActivityGraph” constitutes a doubly linked and the candidates within a pool a singly linked list. The graphical illustration and textual output for “Test02” are shown in figure 4. Each single or candidate activity corresponds to a circle or number triple (id, benefit and cost). The single paths differ from the five route-files of the library only by their indentation. The reason is that the thereto needed spaces would grow quadratically with the nodes. Thus, a further improvement would be a proper encoding of activities when writing them to a file. Although this greatly reduces a file’s size and processing time, such an operation always causes a non-neglectable overhead and should be avoided whenever possible. This can be demonstrated by turning the printing option on/off and comparing the runtime difference.



```
(0, 101, 0)
    (1, 102, 1)  (2, 103, 2)  (3, 104, 3)  (4, 105, 4)  (5, 106, 5)
        (6, 107, 6)  (7, 108, 7)  (8, 109, 8)  (9, 110, 9)  (10, 111, 10)
            (11, 112, 11)  (12, 113, 12)  (13, 114, 13)  (14, 115, 14)  (15, 116, 15)
        (16, 117, 16)
            (11, 112, 11)  (12, 113, 12)  (13, 114, 13)  (14, 115, 14)  (15, 116, 15)
    (17, 118, 17)
        (16, 117, 16)
            (11, 112, 11)  (12, 113, 12)  (13, 114, 13)  (14, 115, 14)  (15, 116, 15)
        (18, 119, 18)  (19, 120, 19)  (20, 121, 20)  (21, 122, 21)  (22, 123, 22)
            (23, 124, 23)  (24, 125, 24)  (25, 126, 25)  (26, 127, 26)  (27, 128, 27)
    (28, 129, 28)  (29, 130, 29)  (30, 131, 30)  (31, 132, 31)  (32, 133, 32)
        (33, 134, 33)
            (34, 135, 34)
```

Figure 4: Graphical and textual representation of an activity graph. Assuming $C_{max} = 35$, the optimal path is marked red.

The inorder traversal of the graph for reading its content into the library (method "BuildTestGraph") is discussed in section 4.4. In contrast to typing each insertion statement manually, "Test05" provides a mean for an automatic construction as visible in listing 9. Altering the upper node limit in the for-loop enables a variable output. This however renders a detailed check on every element of the array impossible.

```
[Test]
public void Test05()
{
    try
    {
        op.RefreshOperator(70, 100, true);


        ActivityGraph graph = new ActivityGraph();

        for (int i = 1; i <= 25; i++)
        {
            if ((i + 1) % 6 == 0)
            {
                graph.GoBack((2 * i) % 10);
            }
            else if (i % 3 == 0)
            {
                graph.AddNext(new Activity(100 * i));
                for (int j = 1; j <= i % 5; j++)
                {
                    graph.AddCandidate(new Activity(100 * i + j, 100 + i + j + (i
                        * j) % (i + j), (i * j) % (i + j)));
                }
            }
            else
```

```
                {
                    // <j> = 2
                    graph.AddNext(new Activity(i, 100 + i + 2 + (i * 2) % (i + 2), (i
                        * 2) % (i + 2)));
                }
            }


            graph.PrintGraph("mi05graph.txt");

            BuildTestGraph(graph.Start);
        }
        catch (Exception e)
        {
            Assert.Fail(e.Message);
        }
    }
```

Listing 9: Constructing an activity graph automatically

True dynamic test instances are a consequence of using random numbers (although they don't have to be truly random but different for each run). "Test06" and "Test07" produce a path and a whole tree of activities varying in their kind and number. They are refined by "Test01" and "Test02" of "InsertActivityLITest.cs" to manage larger inputs and to make trees also change in their structure in addition. The latter is accomplished in "Test02" by filling an array "values" of n (nodes per route) entries with the results of the formula below. Therefore, the probability of branching increases with the actual length of the path (h) because a random number $< n^2$ is compared with $values[h]$. "Test02" also affirms that a best known path exists and that it is not fixed at a certain place in the graph.

$$values[h] = \begin{cases} -1 & h = 0 \\ \lfloor 2h + \frac{h^4}{(n-1)^2} \rfloor & 0 < h < n \end{cases} \qquad 4.12$$

The functionality can be verified by looking for a contradiction concerning the assured quality of a solution. For that purpose, the relative error to the optimum, whose constituents with fixed benefit and cost values have been spread onto an arbitrary route, is considered. The latter has an overall benefit of $150n$ and total costs of $50n$, because a variable initialised with -1 (if n is even) or 50 (if *n* is odd) initiates two successive activities (except the first and second if *n* is odd) on that best path to always sum up to $b_1 + b_2 = 300$ and $c_1 + c_2 = 100$.

Furthermore, the tests of category "li" do not only contribute to a high test coverage but also serve for determining the runtime function (5.34). The codomains for singles, pools and candidates are configurable in "ReturnResultLITest.cs" as displayed in listing 10. For "Test01", their lower/upper bound and increment can be specified appropriately with 5.34 so that the thereby resulting number of test instances terminate in a reasonable time. NODES_LOWER, NODES_UPPER, NODESPR_LOWER and NODESPR_UPPER constitute additional guidelines for the graph composition in "Test02". The remaining constants belong to "Test03" whose task is to execute several (defined by RUNS) tests with a static amount of single, pool and candidate activities.

```
public const int SINGLES_LOWER = 0;//9900000

public const int SINGLES_UPPER = 20;//9900000

// value (> 0) for incrementing number of singles
public const int SINGLES_INC = 1;//9900

public const int SINGLES_MAX = 9900000;

public const int POOLS_LOWER = 0;//1000

public const int POOLS_UPPER = 20;//1000

// value (> 0) for incrementing number of pools
public const int POOLS_INC = 1;

public const int POOLS_MAX = 1000;

public const int CANDIDATES_LOWER = 0;//100

public const int CANDIDATES_UPPER = 20;//100

// value (> 0) for incrementing number of candidates
public const int CANDIDATES_INC = 1;

public const int CANDIDATES_MAX = 100;

public const int NODES_LOWER = 40;

public const int NODES_UPPER = 50;

public const int NODESPR_LOWER = 0;

public const int NODESPR_UPPER = 10;

public const int RUNS = 100;
```

Listing 10: Example settings for random tests

Moreover, “Test03” is suited best for performance tests. If an average runtime is to be determined for the maximum load configuration, the values actually commented out should be used. Secondly, printing has to be turned off. As for “Test01” and “Test02”, the output is written to the corresponding excel file.

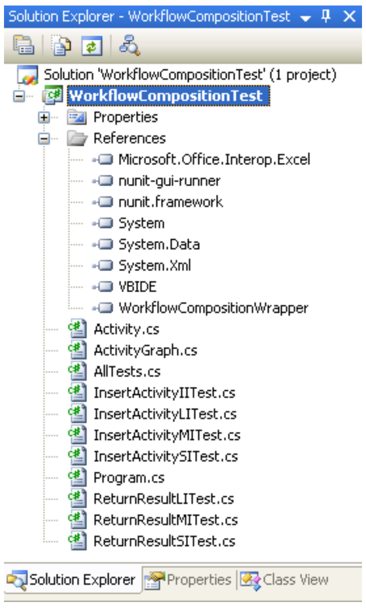
4.4 Usage

This final section describes how the library’s functionality can be harnessed by other applications. The API-functions (i.e. the wrapper) have been introduced previously, here the focus lies in the inorder insertion of activities for their subsequent evaluation (on-the-fly). Furthermore, an activity diagram will show the life cycle of an operator instance. Finally, important things to note when using the wrapper are outlined.

To get the tests work, the setup of the test project shall be discussed first. All of the software was developed in Microsoft Visual Studio 2005. Within WorkflowCompositionLibrary the following dependencies have to be selected:

1. Microsoft Excel 12.0 Object Library
2. nunit.framework
3. nunit-gui-runner
4. WorkflowCompositionWrapper

The first import implicates that version 2007 of Microsoft Excel is installed. NUnit 2.4.3 which is freely available provides nunit.framework.dll and nunit-gui-runner.dll. The last reference together with the unmanaged WorkflowCompositionLibrary.dll (which is instead copied into “bin/Debug”) has to be present in any application that makes use of the program. Once these resources have been added and a refresh has eventually been performed, the folder “References” in the “Solution Explorer” should be in accordance with the one in figure 5. Invoking “Clean Solution”, “Build Solution” and “Start Debugging” or “Start Without Debugging” in sequence will launch the graphical interface of NUnit where the tests that should be run, can be specified.



```
Solution Explorer - WorkflowCompositionTest
Solution 'WorkflowCompositionTest' (1 project)
  WorkflowCompositionTest
    Properties
    References
      Microsoft.Office.Interop.Excel
      nunit-gui-runner
      nunit.framework
      System
      System.Data
      System.Xml
      VBIDE
      WorkflowCompositionWrapper
    Activity.cs
    ActivityGraph.cs
    AllTests.cs
    InsertActivityIITest.cs
    InsertActivityLITest.cs
    InsertActivityMITest.cs
    InsertActivitySITest.cs
    Program.cs
    ReturnResultLITest.cs
    ReturnResultMITest.cs
    ReturnResultSITest.cs
Solution Explorer | Properties | Class View
```

Figure 5: References of WorkflowCompositionTest

After WorkflowCompositionLibrary.dll and WorkflowCompositionWrapper.dll have been included and referenced in a project, an inorder-insertion algorithm needs to be applied for efficiency reason. Method “BuildTestGraph” in “InsertActivityMITest.cs” listed below places one at the user’s disposal. Simply “Activity” and “ActivityGraph” have to be replaced with the appropriate types in the particular application.

```
private void BuildTestGraph(Activity activity)
{
    int i;
    if (activity != null)
    {
        if (activity.Pool)
        {
            op.InsertPoolActivity();
            if ((tmp = activity.Candidate) != null)
            {
                // iterate over the candidates
                while (tmp != null)
                {
                    op.InsertCandidateActivity(tmp.Id, tmp.Benefit, tmp.Cost);
                    tmp = tmp.Candidate;
                }
            }
        }
        else
        {
            op.InsertSingleActivity(activity.Id, activity.Benefit, activity.Cost);
        }

        if (activity.Next != null)
        {
            // iterate over the children
            for (i = 0; i < activity.Next.Count; i++)
            {
                BuildTestGraph((Activity)activity.Next[i]);
            }
        }
    }
    op.InsertBackStatement(1);    // one step back
}
```

Listing 11: Inorder insertion of activities

Once an optimal workflow has been successfully composed, the operator can either be destroyed or preserved for a later update. Since an update constitutes a re-evaluation of a subgraph altered at runtime (structure and/or benefit/cost values of activities) it is also possible to get a new (or preserved) instance from a dispatcher component in an upper architectural tier. Figure 6 clarifies this circumstance with the life cycle of an operator instance.

At the end of this chapter, some useful hints for a correct and efficient use of the software are given.

- An activity's id can be negative but it has to be unique. Otherwise, the result might be ambiguous.
- All benefit (b) and cost values (c) have to be chosen from their specified codomains. Internal overflows caused when the sum of benefits and/or costs exceeds the four byte "int" number range have to be avoided.
- Ordering candidates ascending by b/c increases the number of iterations (high runtime per route).

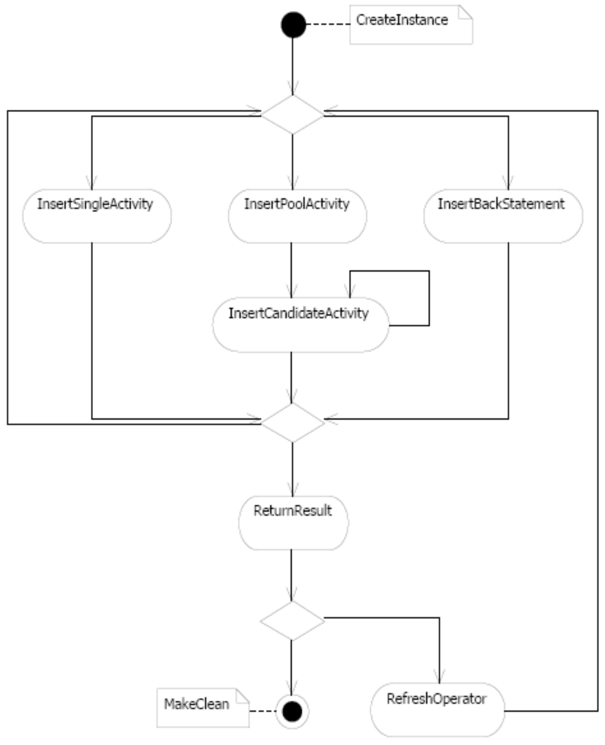


Figure 6: Common use of an operator instance

- Sorting candidates descending by b/c reduces the number of iterations (low runtime per route).
- Evaluating two workflows as one is possible by inserting a common root (dummy activity e.g. an empty pool).
- Routes, that likely contain the optimal selection (e.g. longest paths) should be read in first to skip simplex-calls for subsequent ones (low runtime per workflow).
- The result array is either null or it holds at least two elements. The first entry matches the overall benefit value of the solution found while the others form the ids in the sequence of insertion.

5 Evaluation

This chapter focuses on three aspects characterising the library: Exactness, memory and runtime. The first section quantifies the correctness of a found solution by setting an upper limit on the relative error. Secondly, the program's use of memory resources is discussed. A simple function shows the dependency between user defined input variables and occupied storage. The next to last section is dedicated to close examinations of the implemented algorithm's runtime behaviour. Finally, using the three main equations for the library's configuration is demonstrated with an example.

Appendix C contains statistical data associated exclusively with this chapter. Table 1 presents important characteristics about the computer, the tests were performed on. All values have been rounded to three decimal places for the given magnitude (in square brackets).

Processor	name	Intel Core Duo T2500 / 2 GHz
	multi-core technology	Dual-Core
	data bus speed	667 MHz
RAM	technology	DDR II SDRAM - 533 MHz
	quantity	1 GB
Cache	type	L2
	quantity	2 MB

Table 1: Performance features of the test system

5.1 Exactness

Within this section, the degree of precision concerning the algorithm's proposed solution is discussed. For that purpose, the relative error f is taken as quantifiable measure:

$$f = \frac{B_{opt} - B_l}{B_l} \tag{5.1}$$

B_l constitutes the benefit value of the chosen route, i.e. all benefits of the activities added up to form the solution. The latter corresponds to the linear programming relaxation rounded down. Rounding is necessary in the heuristic, if two candidates j_1 and j_2 have been selected for pool i, otherwise an exact solution has already been found ($f = 0$). It is accomplished by choosing the candidate with the smaller cost value because the other one would make the route's costs exceed the predefined limit. Assuming j_1 causes a valid result then B_l is equivalent to:

$$B_l = b_{ij_1} + \sum_{k=1,k\neq i}^{n} b_k \tag{5.2}$$

Variable k iterates over the selected activities to get representatives for all nodes n (number of single and pool activities) of this route. Just as B_{opt} denotes the sum of benefits in the optimal solution, so does B_s for the linear programming relaxation and B_u for the latter rounded up:

$$B_s = b_{ij_1} x_{ij_1} + b_{ij_2} x_{ij_2} + \sum_{k=1,k\neq i}^{n} b_k = b_{ij_1} + \Delta b_{ij} x_{ij_2} + \sum_{k=1,k\neq i}^{n} b_k = \Delta b_{ij} x_{ij_2} + B_l \tag{5.3}$$

$$B_u = b_{ij_2} + \sum_{k=1,k\neq i}^{n} b_k = \Delta b_{ij} + B_l \tag{5.4}$$

Here, the relation $x_{ij_2} = 1 - x_{ij_1}$ has been used. If $f \neq 0$, the following inequalities are valid:

$$B_l < B_{opt} \leq B_s < B_u \tag{5.5}$$

$B_{opt} = B_s$ indicates that an integer solution of which the sum of benefits is B_s too, does exist. Subtracting B_l from every term allows a proper estimation of $B_{opt} - B_l$. Since the difference between any two benefit values does not exceed a specified limit (b_{min}), $B_{opt} - B_l < b_{min}$ holds for every solution of the heuristic (due to $x_{ij_2} < 1$). The relative error therefore approaches 0 asymptotically for an increasing number of nodes:

$$f < \frac{b_{min}}{b_{ij_1} + \sum_{k=1,k\neq i}^{n} b_k} \leq \frac{b_{min}}{nb_{min}} = \frac{1}{n} \tag{5.6}$$

The more nodes a route has, the higher the relative closeness (to be defined as $1-f$) gets. It's already above 90% when evaluating a path with ten nodes. However, the worst case concerning exactness occurs if there are only two pools. In the smallest problem instance, each of them holds two candidates. Choosing $B_l = b_{12} + b_{21}$ and $B_u = b_{12} + b_{22}$ then $B_s = b_{12} + b_{21}x_{21} + b_{22}x_{22}$ and $B_{opt} = b_{11} + b_{22}$. Equation 5.5 comprises all constraints when maximising $B_{opt} - B_l$ and minimising B_l for f:

$$b_{12} + b_{21} < b_{11} + b_{22} \leq b_{12} + b_{21}x_{21} + b_{22}x_{22} < b_{12} + b_{22} \tag{5.7}$$

$$0 < b_{11} + b_{22} - b_{12} - b_{21} \leq (b_{22} - b_{21})x_{22} < b_{22} - b_{21} \tag{5.8}$$

$B_{opt} - B_l$ is highest for $b_{22} = 2b_{min}$, $b_{21} = b_{min}$ and $x_{22} = 1 - \varepsilon_0$ (i.e. close to 1). B_l would be lowest when setting b_{12} to b_{min} but to the contradiction of $B_{opt} < B_u$. So $b_{11} = b_{min}$ and $b_{12} = b_{min} + \delta_0$ with $\delta_0 \in \mathbb{N}$ as small as possible. The maximum relative error can now be written as:

$$f_{max} = \frac{b_{min} - \delta_0}{2b_{min} + \delta_0} \tag{5.9}$$

What is the smallest value possible for δ_0? Considering the total costs of the linear programming relaxation which coincide with the predefined cost limit C_{max}, ε_0 equals the following expression:

$$\varepsilon_0 = \frac{c_{12} + c_{22} - C_{max}}{c_{22} - c_{21}} \tag{5.10}$$

The formula for ε_0 yields a lower bound of $1/C_{max}$. An upper bound results from 5.8. The variables δ_0 and ε_0 are therefore related as below:

$$\frac{1}{C_{max}} \leq \varepsilon_0 \leq \frac{\delta_0}{b_{min}} \tag{5.11}$$

Setting δ_0 and ε_0 to the smallest values possible (i.e. $\lceil b_{min}/C_{max}\rceil$ and $1/C_{max}$) leads to $c_{11}=0$, $c_{12}=1$, $c_{21}=0$ and $c_{22}=C_{max}$. Furthermore, f_{max} is determined by:

$$f_{max} = \frac{b_{min} - \lceil b_{min}/C_{max}\rceil}{2b_{min} + \lceil b_{min}/C_{max}\rceil} \tag{5.12}$$

As a concrete problem instance with typical test values $b_{min}=100$ and $C_{max}=100$, the worst case situation concerning exactness has the mathematical form:

$$\begin{array}{rrcrcrcrcl}
max & 100x_{11} & + & 101x_{12} & + & 100x_{21} & + & 200x_{22} & & \\
s.t. & & & x_{12} & + & & + & 100x_{22} & \leq & 100 \\
 & x_{11} & + & x_{12} & & & & & = & 1 \\
 & & & & & x_{21} & + & x_{22} & = & 1 \\
 & x_{11} & , & x_{12} & , & x_{21} & , & x_{22} & \in & \{0,1\}
\end{array} \tag{5.13}$$

With $\varepsilon_0=0.01$ the optimal solution found by the simplex algorithm amounts to $x_{11}=0$, $x_{12}=1$, $x_{21}=0.01$ and $x_{22}=0.99$ so that $B_s=300$. The heuristic in turn sets $x_{21}=1$ and $x_{22}=0$ to propose a valid solution. The divergence is highest because another optimal solution $x_{11}=1$, $x_{12}=0$, $x_{21=0}$, $x_{22}=1$ with $B_{opt}=B_s=300$ exists: $f_{max}\approx 0.49$.

The prevention of low correctness ought to be paid attention to in a future release. As the relative error has an impact only on workflows with short routes, solving a small program exactly e.g. with branch-and-bound seems to be suited best. But if the number of candidate activities happens to exceed the nodes by several orders of magnitude another approximation scheme will have to be applied.

5.2 Memory

Memory allocation occurs only once at the beginning in method “init” of “export.c”. This can be seen as a consequence of the experienced performance loss in case of dynamically allocating storage. The disadvantage of statically using resources is leveraged by giving the calling application the opportunity to decide on the maximum amount of MB for processing a route. The latter turns out to be important when figuring out the number of single, pool and candidate activities so that everything fits into main memory or does not exceed the storage capabilities of the device itself.

Hence, a relation between the specified number of activities and their associated need of storage, expressed by $f_S(S,P,C)$ is needed. S, P and C do not necessarily coincide with $MAX\_ACTIVITIES - MAX\_POOLS * MAX\_CANDIDATES$ (the maximum number of singles), MAX_POOLS and MAX_CANDIDATES. The latter are typically adapted only once for the computer hosting the library while the former represents a client's actual request (i.e. the current mode of operation) in a web service-like scenario.

As previously mentioned, the malloc-statements are concentrated in the initialisation process. An exception is thrown if only a single call to “malloc” fails. This all-or-nothing strategy relieves other methods being called more frequently from this costly process. The code snippet for memory allocation is printed in listing 12.

```
if ((activities = malloc(num_activities * ACTIVITY_SIZE)) == NULL ||
    (rids = malloc((2 + num_activities) * sizeof(rids[0]))) == NULL ||
    (pools = malloc(num_pools * POOL_SIZE)) == NULL ||
    (basis = malloc(2 * sizeof(basis[0]))) == NULL ||
    (basis[0] = malloc(2 * (1 + num_pools) * sizeof(basis[0][0]))) == NULL ||
    (xB = malloc((1 + num_pools) * sizeof(xB[0]))) == NULL ||
    (y = malloc((1 + num_pools) * sizeof(y[0]))) == NULL ||
    (d = malloc((1 + num_pools) * sizeof(d[0]))) == NULL ||
    (sB = malloc((1 + num_pools) * sizeof(sB[0]))) == NULL ||
    (best = malloc(num_pools * sizeof(best[0]))) == NULL ||
    (fname = malloc(LENGTH * sizeof(fname[0]))) == NULL ||
    (routetos = malloc((1 + NUM_DIGITS) * sizeof(routetos[0]))) == NULL)
{
    leave(ERR_MEMORY);
}
```

Listing 12: Memory allocation

The size of all common fields and structures can be found in appendix B. Adding up all parts leads to the function given below:

$$f_S(S,P,C) = 16S + 56P + 16CP + 83 \qquad 5.14$$

This equation states the total amount of allocated bytes depending on the defined number of activities. It enables the configuration of the library regarding storage issues. Section 5.4 shows an example for a detailed customisation of WorkflowCompositionLibrary for additional information about execution time and nodes per route.

5.3 Runtime

The following sections deal with theoretical and experimental runtime determination. “Test01” in “ReturnResultLiTest.cs” happens to be most suitable to accomplish this for a single route. The first configuration helps in assigning concrete values to all coefficients in the route’s runtime function. Their correctness and possible improvements are examined thereafter. Finally, the algorithm’s runtime behaviour in a real abstract workflow with completely different paths is discussed.

5.3.1 Runtime Equation of a Route

Runtime generally depends upon three variables: the number of single activities (s), the amount of pool activities (p) and the candidate activities (c). It can be savely assumed that c is constant for every pool on a certain path.

Such as the library’s functionality is basically made up of read-in process and optimisation procedure, so the runtime is. If printing is turned off, the overhead of the former is caused by the methods “insertsa”, “insertpa”, “insertca” and “reset”. Table 2 shows the function’s runtime contribution when processing the input.

Method	**Complexity**
insertsa	$\Theta(s)$
insertpa	$\Theta(p)$
insertca	$\Theta(cp)$
reset	$\Theta(s+p)$

Table 2: Input processing methods and their complexity per route

The extra invocation of “insertpa” initiates the execution of the algorithm. The latter is composed of several parts contributing to the overhead. Table 3 shows the situation at a glance for the steering function “trigger”.

trigger			$\Omega(p), O(s+p+p^2 log(cp)+cp^2 log(cp))$
	shortcut		$\Omega(p), O(c+p)$
	simplex		$\Omega(p+cp), O(p+p^2 log(cp)+cp^2 log(cp))$
		step0	$\Theta(p)$
		step234	$\Theta(p+cp)$
		refactor	$\Theta(p)$
	eval		$\Theta(p)$
	fillr		$\Theta(s+p)$

Table 3: Algorithmic break down of complexity

The estimation is straight-forward for most of the functions listed. However, “simplex” deserves a bit more attention. As stated in the previous chapter, the simplex algorithm consists of five steps that are continuously repeated. An additional method “step0” exclusively serves for initialisation purposes. The decisive question “How many iterations can be expected for the while-loop at the beginning?” cannot be settled easily on a theoretical basis. According to studies conducted in this area (Chvátal [Chv83]), the number of iterations increases proportionally to the number of constraints $(p+1)$ and the logarithm of the number of real variables (cp).

This circumstance turns out to be true for most problems and therefore constitutes the simplex method's wide spread. Unfortunately, examples do exist where the number of iterations is of an exponential size (Klee et al. [KM72]). However, this worst case scenario seems to be very rare, otherwise the above mentioned studies would have shown diverging results. Therefore, $p\,log(cp)$ has been chosen as upper bound function for the total amount of iterations in "simplex".

If the start solution is already close to the optimum, only a constant number of iterations is necessary. Because of such cases, its lower bound has been set on $\Omega(1)$. In general, reducing the number of iterations is not only possible by finding a good start solution but also when judging candidates for entering the basis (step 2) by some good strategy. Chvátal [Chv83] mentions in this context the "Devex" and "steepest edge" criterion with a possible overhead reduction of more than 50%.

The time an iteration takes can be derived from each single step of the algorithm and "refactor". The latter is linear to p although in practice the period for refactoring can be chosen arbitrarily large (to the disadvantage of accuracy though). Accounting all relevant parts, the complexity per iteration can be estimated as $\Theta(p+cp)$. Lower and upper bound functions of the latter explain Ω- and O-expression for describing the simplex's overhead. Function "trigger" requires a minimum of resources if the start solution is infeasible or below a predetermined bound. If the input allows no shortcut, then "simplex" together with "fillr" contributes to its running time.

The reason for not merging unequal terms in Ω-, Θ- and O-notation as it would have been possible for e.g. $O(p+p^2log(cp)+cp^2log(cp))=O(cp^2log(cp))$ in "simplex" becomes evident when setting up a more accurate runtime-equation for a route:

$$f_R(s,p,c)=\alpha s+\beta p+\gamma cp+\delta p^2ln(cp)+\varepsilon cp^2ln(cp)+\zeta \qquad 5.15$$

Here, the number of candidates and pools is assumed to be positive. To consider also 0-values, the expression could be replaced with $ln(1+\iota cp)$. However, the additional coefficient constitutes a numerical problem. To get rid of it, the term is replaced by 0 for $p, c = 0$ and $ln(cp)$ otherwise with $ln(\imath)$ being subsumed by ζ.

$$ln(1 + \imath cp) \overset{cp \gg 1}{\approx} ln(\imath cp) = ln(cp) + ln(\imath) \qquad 5.16$$

5.3.2 Coefficient Determination

A first series of tests aims at the assignment of good estimations to all coefficients of the function. The arithmetic mean is then chosen as good approximation though the true values remain unknown for a finite number of tests.

If multiple measurements are performed keeping s, p and c constant then the output data r will follow a normal distribution. This can be seen as a consequence of stochastic independence among r (true randomness), described by the central limit theorem (Viertl [Vie03]). Systematic errors have been avoided. The equation below represents the Gaussian distribution of r.

$$p(r) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(r - f_R(s_0, p_0, c_0))^2}{2\sigma^2}} \qquad 5.17$$

The probability of getting $f_R(s_i, p_i, c_i)$ out of m triples (s_i, p_i, c_i) by m independent measurements (Demtroeder [Dem03]) can be written as:

$$P(r_1, \ldots, r_m) = \prod_{i=1}^{m} p(r_i) \qquad 5.18$$

$$P(r_1, \ldots, r_m) = \left(\frac{1}{2\pi\sigma^2}\right)^{\frac{m}{2}} e^{-\frac{\sum_{i=1}^{m}(r_i - f_R(s_i, p_i, c_i))^2}{2\sigma^2}} \qquad 5.19$$

Its largest value indicates that the best coefficients have been found for the input test data. As expected, the method of least squares is a direct consequence of maximising $P(r_1, \ldots, r_m)$:

$$\sum_{i=1}^{m}(r_i - f_R(s_i, p_i, c_i))^2 \rightarrow min \qquad 5.20$$

To keep the syntax short, $f_R(s,p,c)$ is put into vector notation:

$$f_R(s,p,c) = \vec{x}^T\vec{\theta} \qquad \vec{x}^T := \begin{pmatrix} s & p & cp & p^2 ln(cp) & cp^2 ln(cp) & 1 \end{pmatrix}, \quad \vec{\theta} := \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \\ \varepsilon \\ \zeta \end{pmatrix} \tag{5.21}$$

Calculating the minimum means setting all partial derivatives to 0. By using the nabla operator ∇, this condition is focused in the following equation:

$$\nabla\left(\sum_{i=1}^{m}(r_i - \vec{x}_i^T\vec{\theta})^2\right) = \vec{0} \qquad \nabla = \begin{pmatrix} \frac{\partial}{\partial\alpha} \\ \frac{\partial}{\partial\beta} \\ \frac{\partial}{\partial\gamma} \\ \frac{\partial}{\partial\delta} \\ \frac{\partial}{\partial\varepsilon} \\ \frac{\partial}{\partial\zeta} \end{pmatrix} \tag{5.22}$$

$$\sum_{i=1}^{m}\nabla(r_i - \vec{x}_i^T\vec{\theta})^2 = \vec{0} \tag{5.23}$$

$$\sum_{i=1}^{m}2(r_i - \vec{x}_i^T\vec{\theta})(-\nabla(\vec{x}_i^T\vec{\theta})) = \vec{0} \tag{5.24}$$

$$\sum_{i=1}^{m}(r_i - \vec{x}_i^T\vec{\theta})\vec{x}_i = \vec{0} \tag{5.25}$$

$$\sum_{i=1}^{m}(\vec{x}_i^T\vec{\theta})\vec{x}_i = \sum_{i=1}^{m}r_i\vec{x}_i \tag{5.26}$$

To extract the parameter vector $\vec{\theta}$ out of $(\vec{x}_i^T\vec{\theta})\vec{x}_i$ the subsequent equivalence is used.

$$(\vec{x}_i^T\vec{\theta})\vec{x}_i = \begin{pmatrix} s_i\vec{x}_i^T \\ p_i\vec{x}_i^T \\ c_ip_i\vec{x}_i^T \\ p_i^2 ln(c_ip_i)\vec{x}_i^T \\ c_ip_i^2 ln(c_ip_i)\vec{x}_i^T \\ \vec{x}_i^T \end{pmatrix}\vec{\theta} \tag{5.27}$$

The linear system of equations to solve for $\vec{\theta}$ is conform to $\boldsymbol{A}\vec{\theta} = \vec{b}$ with symmetric matrix $\boldsymbol{A}$ and $\vec{b}$ as follows.

$$\boldsymbol{A} := \begin{pmatrix} \sum_{i=1}^{m} s_i^2 & \sum_{i=1}^{m} s_i p_i & \sum_{i=1}^{m} s_i c_i p_i & \sum_{i=1}^{m} s_i p_i^2 ln(c_i p_i) & \sum_{i=1}^{m} s_i c_i p_i^2 ln(c_i p_i) & \sum_{i=1}^{m} s_i \\ a_{12} & \sum_{i=1}^{m} p_i^2 & \sum_{i=1}^{m} c_i p_i^2 & \sum_{i=1}^{m} p_i^3 ln(c_i p_i) & \sum_{i=1}^{m} c_i p_i^3 ln(c_i p_i) & \sum_{i=1}^{m} p_i \\ a_{13} & a_{23} & \sum_{i=1}^{m} c_i^2 p_i^2 & \sum_{i=1}^{m} c_i p_i^3 ln(c_i p_i) & \sum_{i=1}^{m} c_i^2 p_i^3 ln(c_i p_i) & \sum_{i=1}^{m} c_i p_i \\ a_{14} & a_{24} & a_{34} & \sum_{i=1}^{m} p_i^4 ln^2(c_i p_i) & \sum_{i=1}^{m} c_i p_i^4 ln^2(c_i p_i) & \sum_{i=1}^{m} p_i^2 ln(c_i p_i) \\ a_{15} & a_{25} & a_{35} & a_{45} & \sum_{i=1}^{m} c_i^2 p_i^4 ln^2(c_i p_i) & \sum_{i=1}^{m} c_i p_i^2 ln(c_i p_i) \\ a_{16} & a_{26} & a_{36} & a_{46} & a_{56} & m \end{pmatrix} \quad 5.28$$

$$\vec{b} := \begin{pmatrix} \sum_{i=1}^{m} r_i s_i \\ \sum_{i=1}^{m} r_i p_i \\ \sum_{i=1}^{m} r_i c_i p_i \\ \sum_{i=1}^{m} r_i p_i^2 ln(c_i p_i) \\ \sum_{i=1}^{m} r_i c_i p_i^2 ln(c_i p_i) \\ \sum_{i=1}^{m} r_i \end{pmatrix} \quad 5.29$$

The general function 5.15 includes a linear dependency of s. If $p = 0$ and therefore also $c = 0$, a linear regression model to obtain formulas for α and ζ is taken into consideration. However, this job is also done by calculating $\boldsymbol{A}\vec{\theta} = \vec{b}$ for that special case.

$$\begin{pmatrix} \sum_{i=1}^{m} s_i^2 & \sum_{i=1}^{m} s_i \\ \sum_{i=1}^{m} s_i & m \end{pmatrix} \cdot \begin{pmatrix} \alpha \\ \zeta \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{m} r_i s_i \\ \sum_{i=1}^{m} r_i \end{pmatrix} \quad 5.30$$

Solving this system leads to equations for α and ζ. Comparing them with Bronstein et al. [BS85] for instance, shows accordance.

$$\alpha = \frac{m\sum_{i=1}^{m} r_i s_i - \sum_{i=1}^{m} r_i \sum_{i=1}^{m} s_i}{m\sum_{i=1}^{m} s_i^2 - \left(\sum_{i=1}^{m} s_i\right)^2} \quad 5.31$$

$$\zeta = \frac{\sum_{i=1}^{m} r_i \sum_{i=1}^{m} s_i^2 - \sum_{i=1}^{m} r_i s_i \sum_{i=1}^{m} s_i}{m\sum_{i=1}^{m} s_i^2 - \left(\sum_{i=1}^{m} s_i\right)^2} \quad 5.32$$

Unfortunately, finding formulas for all variables in the general case is hardly manageable. Hence, solving this system of linear equations numerically remains an alternative. Modern spreadsheets offer the possibility of keeping the solution finding process reusable.

Characteristic data about the test environment can be found in appendix C. As computers may differ heavily, the question about meaningfulness concerning coefficient determination on a specific machine arises. Anyway, repeating one of the test suites in this appendix and comparing the average values, leads to the right proportionality constant to multiply the runtime function with.

To summarise, ten tests have been run to get significant values for each parameter. The codomain of s, p and c has been fixed as follows to get good representatives out of all possible varible-value combinations. Table 4 shows the calculated parameters for the tests performed. The abbreviation $l[i]u$ denotes lower and upper bound with the increment in square brackets. Due to these settings, the number of measurements amounts to 330.

Concluding this section, the execution time (in 10^{-7}seconds) per route is expressed by:

$$f_R(s,p,c) = 2.42s + 18p + 8.8cp + 0.019p^2 ln(cp) + 0.010cp^2 ln(cp) + 96000 \tag{5.33}$$

In practice, most of the data are dispersed around this function value in the average confidence interval [-0.05s, +0.05s]. Fortunately δ and ε, the parameters of the terms most decisive for the overall runtime, are small compared to the others. That circumstance could be explained by the little influence of $ln(cp)$.

5.3.3 Adjusting Parameters

This section describes case studies with respect to parameter tuning techniques. They do not only lead to more precise values but also allow a verification of equation 5.33.

Coefficient α can be determined more accurately. However, the performance data in table 4 narrows its value to 2.42E-7$\pm$0.01 thus leaving only the hundredth of a second undetermined (for $s = 10^7$). To verify this estimation, a fine-grained test has been carried out by setting $p, c = 0$ – with the function to analyse evaluating to $f_R(s,0,0) = \alpha s + B$. Generally, $B \neq \zeta$ as there are fewer terms contributing to the runtime in this case. The number of single activities has been kept variable in the same interval but being incremented by 9900 in each run. So $m = 1001$ and α is calculated directly by formula 5.31. As a result, the value has been computed to 2.420E-7 which almost coincides with the mean value. Figure 7 represents this test graphically.

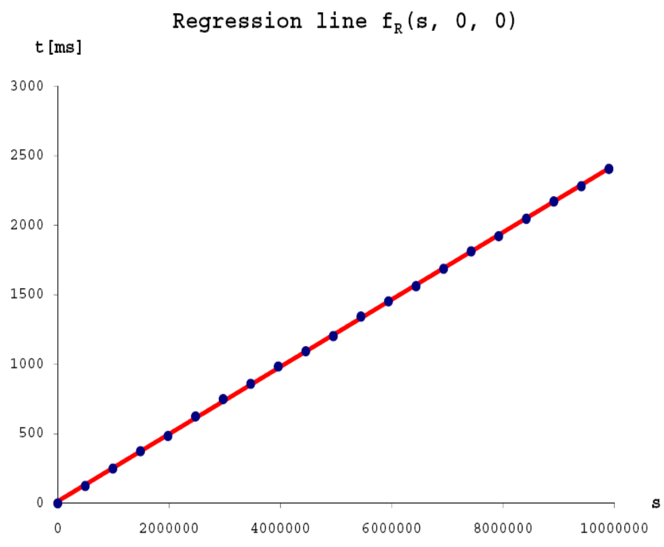


Figure 7: Estimation of α with $f_R(s, 0, 0)$

Parameter tuning for β, γ, δ and ε would be desirable for accurate runtime predictions. Unfortunately, the values in table 4 are quite diverging. This could be a consequence of how candidate activities are selected from a pool (step 2 of the algorithm). To receive more stable values, the mean of another ten tests shall be considered. Setting $s = 0$ for instance, would help to focus on the influence of p and c in the simplex method. As in the first test suite, the values for the number of candidates and pools have been chosen uniformly from their complete codomains. An in-depth analysis is achieved by lowering the loop increment e.g. on 10 for both variables. According to these settings, 1111 measurements are necessary for $f_R(0, p, c) = \beta p + \gamma cp + \delta p^2 ln(cp) + \varepsilon cp^2 ln(cp) + B$, with $B \neq \zeta$ again.

As a representative, the function of test01 is plotted (figure 8). The single measurements have been omitted for clarity reasons. Interestingly, the function values drop below zero for small values of p and c. This turns out to be a consequence of $0 < \iota < 1$ (in the incorrect approximation 5.16 for $cp \not\gg 1$) because $ln(\iota)$ is negative in this interval leading to a negative B.

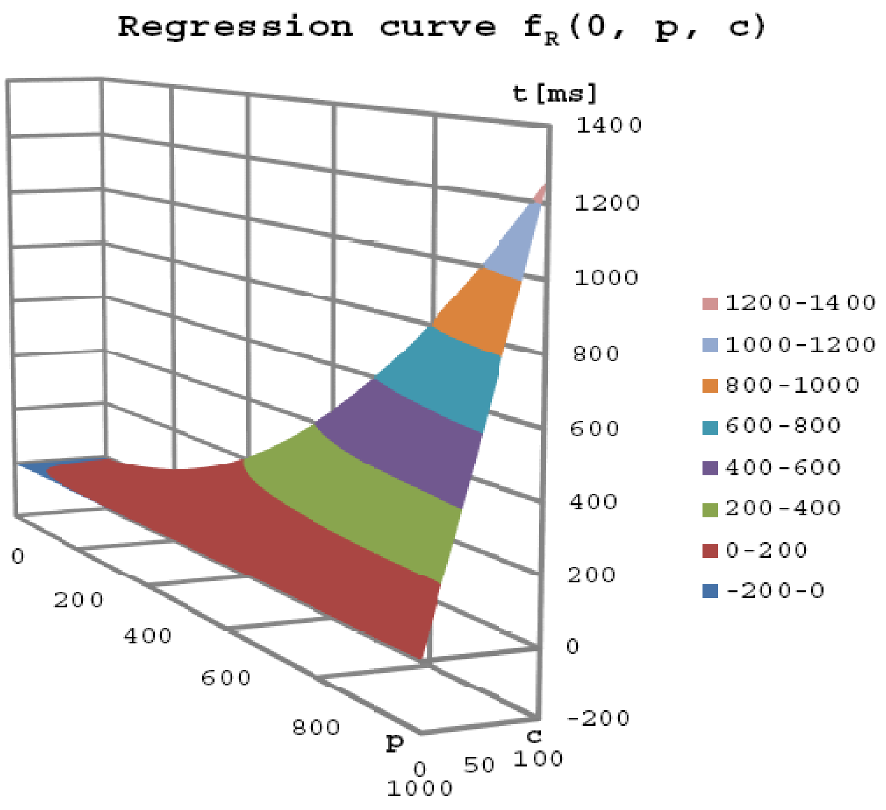


Figure 8: Estimation of β, γ, δ and ε with $f_R(0, p, c)$

Table 5 demonstrates the results of the tests launched. Fortunately, the values are more stable this time, but unexpectedly higher (especially β). Therefore, another series of tests keeps s varying among the tests (i.e. generates cuts of the runtime function) although no dependency on the number of single activities exists. This can be seen in table 6.

Fixing s, p and c allows for a closer determination of ζ. "Test03" assigns random values to these variables (within their appropriate codomains). As far as the number of measurements is concerned, 100 has been chosen. The other parameters have been adapted to the previous results. The outcome can be viewed in table 7.

Unfortunately, ζ varies heavily over two orders of magnitude. However, weighting ζ to the mean execution time $\langle r \rangle$ shortens this interval to 0.04. Moreover, $\zeta/\langle r \rangle$ appears to be independent of s, p and c. Substituting $f_R(s,p,c)$ for $\langle r \rangle$ and dissolving the reflective relationship leads to a ζ-free runtime equation. So every term is multiplied by $1/(1-\kappa)$ with κ being the average of $\zeta/\langle r \rangle$. With $\kappa = 0.03$ and the improved parameter values (table 7) equation 5.33 is updated as follows:

$$f_R(s,p,c) = 2.49s + 38p + 9.5cp + 0.026p^2 ln(cp) + 0.010cp^2 ln(cp) \qquad 5.34$$

Testing this function for the maximum load preconfigured – i.e. 100 000 variables ($p = 1000$, $c = 100$) for the simplex and 9900 000 single activities – yields 3.745[s]. Applying these settings on “Test03” produces the average 3.754[s] – a neglectable difference.

5.3.4 Runtime Estimation for a Workflow

An abstract workflow's execution time depends on the arrangement of its single routes. Candidate activities sorted in ascending order by their benefit values lead to an unnecessary delay. The same applies to paths at a higher level of granularity. The calling application can speed up processing of a workflow when passing probable candidate routes first.

However, the decision which path to bring forward cannot be made easily if they don't differ noticeably in length or in their activities' characteristics. In such cases it's best to rely on the built-in shortcut criteria. The general overhead of a workflow can be written as:

$$f_W(s,p,c)_{act} = \eta f_W(s,p,c)_{exp} \qquad 0 < \eta \leq 1 \qquad 5.35$$

$$f_W(s,p,c)_{act} = \eta \sum_{i=1}^{N} f_R(s,p,c)_i \qquad 5.36$$

The factor η describes the relationship between the workflow's actual and expected processing time. It is lowest if a shortcut is possible for every route and highest in case of evaluating every route. Thus η serves as an indicator for synergy originating from a path's dependency on its predecessors.

5.4 Customisation

The preset maximum number of single, pool and candidate activities occupies about 153MB. Hence, a main storage of 256MB RAM at least is a necessary precondition for the tests and advisable when using the library in common. But how to adjust the library so that it uses only 10MB at a maximum? Another configurational aspect is execution time. Using the program in a web service environment for instance, might impose an upper bound for the net processing time per route of 1 second. Finally, the calling application itself has to cope with routes of a certain length and passes the constraint (not more than 500 000 nodes) to WorkflowCompositionLibrary.

This challenge can be met by recalling the functions for storage and runtime together with the formula for the number of nodes per route:

$$\begin{array}{rcrcrcrcl} S & + & P & & & & & = & 500000 \\ 16S & + & 56P & + & 16CP & + & 83 & = & 10485760 \\ 2.49S & + & 38P & + & 9.5CP & + & (0.01C+0.026)P^2 ln(CP) & = & 10000000 \end{array} \tag{5.37}$$

Substituting S and P in the last equation which is then to be solved numerically results into $MAX\_POOLS = 392$ and $MAX\_CANDIDATES = 393$.

Customising WorkflowCompositionLibrary that way could be problematic, because a solution to the equation system 5.37 does not necessarily exist for all feasible settings. Another strategy consists in fixing the maximal number of single and candidate activities and calculating *MAX_POOLS* from the runtime equation for a given upper time limit.

There are several reasons, why single activities should not be read into the library. Firstly, they do not participate in the evaluation process. For a calling application, the interesting thing is just to get the right candidates out of the pools. Secondly, they occupy most of the storage and contribute remarkably to the processing time – a dynamic update slows down needlessly.

Besides setting $S = 0$, $C = 50$ can be expected in the context of web services. It's smaller of course, if the activities represent tasks to be fulfiled by persons. If the evaluation of a route should not last more than a second, then $MAX\_POOLS = 1267$. This configuration needs only 1MB of memory.

6 Conclusion

Concluding this thesis, some final remarks on the model, its mathematical formulation and the implementation are given. The separation to what should be covered in a next version is discussed afterwards. At the end, the content of this work and the main features of the algorithm are summarised.

6.1 Final Remarks

The approach presented in this work addresses the problematic characteristics of long workflows: combinatorial issues and dynamic aspects. The underlying abstract workflow model keeps the mathematical formulation of the optimisation problem simple as opposed to other proposals reviewed in chapter 2. This is achieved by mapping QoS criteria such as availability, reliability and execution time for each activity to a single benefit and cost value. Mapping functions have been proposed ad hoc, but they may be adjusted to specific needs in consideration of the appropriate codomains. So they provide a flexible mean of adaptation to a variation in the quality of service, instead of modifying the solution finding process.

Moreover, this simple formulation leads to an eased implementation of the revised simplex method. Particularly, the linear equation systems of step 1 and 2 (section 3.2.3) can be solved directly without the need of a factorisation. Additionally, rounding of the relaxation is facilitated, i.e. it's a consequence of at most two fractional variables in just one pool. This is again due to the special form of the coefficient matrix. Finally, even the upper bound on the relative error (the reason for denoting the heuristic as approximative) turns out to be on behalf of the mapping for benefit values.

Although a few points deserve a closer study (section 6.2), the approximation algorithm constitutes a reasonable tradeoff between accuracy and speed for big instances. It remains to explore, if exact methods are similar efficient for justifiable constraints on the user input.

6.2 Future Work

First of all, the quality limit needs to be lifted for routes with e.g. fewer than ten nodes. An exact method would serve best for a low number of candidates per pool. Otherwise, a further heuristic or approximative algorithm would sound promising. A convex hull determination as done by Yang et al. [YTX+07] may be applied. The proposition of finding the optimal solution with branch-and-bound and linear programming relaxations is discussed in section 3.3. However, its suitability for a high amount of pools (and/or candidates therein) has to be examined. Including the user into the decision on the degree of efficiency can be arranged by providing a specifiable time constraint or an explicit query on the type of algorithm to use.

In the next version, the present implementation itself ought to be ameliorated concerning execution time and accuracy. The former can be reduced by decreasing the number of iterations in the simplex method. More precisely, this could be accomplished by sorting the candidates in a pool descending by b/c or changing the existing pivoting strategy for the entering variable. Accuracy can be improved by augmenting the lower bound set by the heuristic: First, the pool with two variables in the basis can be iterated over for the best matching candidate. Second, enforcing a restriction on the cost value of an activity (i.e. $c_{min} \leq c \leq 2c_{min}$) likewise on its benefit, narrows the deviation from the total costs of the optimal solution to less than c_{min}. It would then be possible to check for a better limit by varying the predefined costs (C_{max}) at most $ld(c_{min})$ times. After those changes will have been applied, measurements for the statistic distribution of accuracy will be of a great interest. Minimising the costs for a given solution might increase the lower bound as well. Besides, a secondary cost optimisation, which is currently only implemented in “shortcut” would be attractive for the user as well.

Finally, the software should be adapted technically for an integration into a workflow framework or middleware. To serve any specification, the support for different programming languages has priority. It is achieved by additionally placing wrappers for other object-oriented languages like C++ and Java at the disposal. Platform independence requires to replace Visual C constructs (such as "fopen_s") with those corresponding to the ANSI C standard in general. As for the necessary data exchange, developing an encoding scheme for activities may get appealing with respect to transfer time and size of the data.

6.3 Summary

Beginning with a survey on existing approaches of service composition, a new model of an abstract workflow has been introduced. It provides a simplified view on the structure of the graph and its constituents. Each activity at this lowest architectural tier is assigned a benefit and a cost value. Both parameters are the result of a function, taking high level QoS criteria as input. A formulation of the problem as binary linear program has led to the simplex algorithm for solving its relaxation in the general case. Rounding of the output to a valid solution is facilitated by at most two fractional variables. Furthermore, exact methods have been proposed as a suitable extension. Implementation details of the revised simplex method have been presented thereafter, together with appropriate code excerpts for library, wrapper and test project. Section "Usage" constitutes the setup in other projects and important things to note. In the last part, a thorough evaluation regarding exactness, memory and runtime has been conducted.

The algorithm performs best for paths containing many nodes (n). The relative error is bounded by $1/n$ and the runtime goes almost quadratically to the number of pools for a constant amount of candidates therein. The latter allows dynamic updates where the actual subgraph can be read in on-the-fly. Preferably, the routes which likely comprise the optimal solution should be inserted first. Tuning is foremost possible by ordering the candidates in a pool descending by the ratio b/c. Generally, the library can be customised for a more lightweight use in e.g. mobile phones or web services environments. For that purpose, the import of single activities may be skipped and the maximum of candidates may be fixed to some reasonable value before considering the runtime and memory equation.

A Applications

Example: $max\,\{\vec{b}^T\vec{x} \mid \boldsymbol{C}\vec{x} = \vec{a}, \vec{x} \geq \vec{0}\}$

$$\vec{a} = \begin{pmatrix} 1 & 1 & 100 \end{pmatrix}^T$$

$$\vec{b}^T = \begin{pmatrix} 133 & 135 & 133 & 142 & 142 & 161 & 133 & 146 & 125 & 144 & 136 & 197 & 0 \end{pmatrix}$$

$$\boldsymbol{C} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 50 & 92 & 42 & 66 & 38 & 96 & 62 & 5 & 86 & 47 & 14 & 1 \end{pmatrix}$$

A.1 Standard Simplex Method

−258	0	2	0	9	9	28	8	21	0	19	11	72	0
1	1	$\boxed{1}$	1	1	1	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	1	1	1	1	1	0
95	0	50	92	42	66	38	91	57	0	81	42	9	1

−260	−2	0	−2	7	7	26	8	21	0	19	11	72	0
1	1	1	1	$\boxed{1}$	1	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	1	1	1	1	1	0
45	−50	0	42	−8	16	−12	91	57	0	81	42	9	1

−267	−9	−7	−9	0	0	19	8	21	0	19	11	72	0
1	1	1	1	1	1	$\boxed{1}$	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	1	1	1	1	1	0
53	−42	8	50	0	24	−4	91	57	0	81	42	9	1

−286	−28	−26	−28	−19	−19	0	8	21	0	19	11	72	0
1	1	1	1	1	1	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	1	1	1	1	1	0
57	−38	12	54	4	28	0	$\boxed{91}$	57	0	81	42	9	1

$-291\frac{1}{91}$	$-24\frac{60}{91}$	$-27\frac{5}{91}$	$-32\frac{68}{91}$	$-19\frac{32}{91}$	$-21\frac{6}{13}$	0	0	$15\frac{90}{91}$	0	$11\frac{80}{91}$	$7\frac{4}{13}$	$71\frac{19}{91}$	$-\frac{8}{91}$
1	1	1	1	1	1	1	0	0	0	0	0	0	0
$\frac{34}{91}$	$\frac{38}{91}$	$-\frac{12}{91}$	$-\frac{54}{91}$	$-\frac{4}{91}$	$-\frac{4}{13}$	0	0	$\boxed{\frac{34}{91}}$	1	$\frac{10}{91}$	$\frac{7}{13}$	$\frac{82}{91}$	$-\frac{1}{91}$
$\frac{57}{91}$	$-\frac{38}{91}$	$\frac{12}{91}$	$\frac{54}{91}$	$\frac{4}{91}$	$\frac{4}{13}$	0	1	$\frac{57}{91}$	0	$\frac{81}{91}$	$\frac{6}{13}$	$\frac{9}{91}$	$\frac{1}{91}$

$$\begin{array}{r|rrrrrrrrrrrrr}
-307 & -42\frac{9}{17} & -21\frac{7}{17} & -7\frac{6}{17} & -17\frac{8}{17} & -8\frac{5}{17} & 0 & 0 & 0 & -42\frac{27}{34} & 7\frac{3}{17} & -15\frac{25}{34} & 32\frac{11}{17} & \frac{13}{34} \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1\frac{2}{17} & -\frac{6}{17} & -1\frac{10}{17} & -\frac{2}{17} & -\frac{14}{17} & 0 & 0 & 1 & 2\frac{23}{34} & \frac{5}{17} & 1\frac{15}{34} & 2\frac{7}{17} & -\frac{1}{34} \\
0 & -1\frac{38}{323} & \frac{6}{17} & 1\frac{10}{17} & \frac{2}{17} & \frac{14}{17} & 0 & 1 & 0 & -1\frac{23}{34} & \boxed{\frac{12}{17}} & -\frac{15}{34} & -1\frac{7}{17} & \frac{1}{34}
\end{array}$$

$$\begin{array}{r|rrrrrrrrrrrrr}
-307 & -31\frac{1}{6} & -25 & -23\frac{1}{2} & -18\frac{2}{3} & -16\frac{2}{3} & 0 & -10\frac{1}{6} & 0 & -25\frac{3}{4} & 0 & -11\frac{1}{4} & 47 & \frac{1}{12} \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1\frac{7}{12} & -\frac{1}{2} & -2\frac{1}{4} & -\frac{1}{6} & -1\frac{1}{6} & 0 & -\frac{5}{12} & 1 & 3\frac{3}{8} & 0 & 1\frac{5}{8} & \boxed{3} & -\frac{1}{24} \\
0 & -1\frac{7}{12} & \frac{1}{2} & 2\frac{1}{4} & \frac{1}{6} & 1\frac{1}{6} & 0 & 1\frac{5}{12} & 0 & -2\frac{3}{8} & 1 & -\frac{5}{8} & -2 & \frac{1}{24}
\end{array}$$

$$\begin{array}{r|rrrrrrrrrrrrr}
-322\frac{2}{3} & -55\frac{35}{36} & -17\frac{1}{6} & 11\frac{3}{4} & -16\frac{1}{18} & 1\frac{11}{18} & 0 & -3\frac{23}{36} & -15\frac{2}{3} & -78\frac{5}{8} & 0 & -36\frac{17}{24} & 0 & \frac{53}{72} \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{3} & \frac{19}{36} & -\frac{1}{6} & -\frac{3}{4} & -\frac{1}{18} & -\frac{7}{18} & 0 & -\frac{5}{36} & \frac{1}{3} & 1\frac{1}{8} & 0 & \frac{13}{24} & 1 & -\frac{1}{72} \\
\frac{2}{3} & -\frac{19}{36} & \frac{1}{6} & \boxed{\frac{3}{4}} & \frac{1}{18} & \frac{7}{18} & 0 & 1\frac{5}{36} & \frac{2}{3} & -\frac{1}{8} & 1 & \frac{11}{24} & 0 & \frac{1}{72}
\end{array}$$

$$\begin{array}{r|rrrrrrrrrrrrr}
-333\frac{1}{9} & -47\frac{19}{27} & -19\frac{7}{9} & 0 & -16\frac{25}{27} & -4\frac{13}{27} & 0 & -21\frac{13}{27} & -26\frac{1}{9} & -76\frac{2}{3} & -15\frac{2}{3} & -43\frac{8}{9} & 0 & \frac{14}{27} \\
\hline
\frac{1}{9} & 1\frac{19}{27} & \frac{7}{9} & 0 & \frac{25}{27} & \frac{13}{27} & 1 & -1\frac{14}{27} & -\frac{8}{9} & \frac{1}{6} & -1\frac{1}{3} & -\frac{11}{18} & 0 & -\frac{1}{54} \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
\frac{8}{9} & -\frac{19}{27} & \frac{2}{9} & 1 & \frac{2}{27} & \frac{14}{27} & 0 & 1\frac{14}{27} & \frac{8}{9} & -\frac{1}{6} & 1\frac{1}{3} & \frac{11}{18} & 0 & \boxed{\frac{1}{54}}
\end{array}$$

$$\begin{array}{r|rrrrrrrrrrrrr}
-358 & -28 & -26 & -28 & -19 & -19 & 0 & -64 & -51 & -72 & -53 & -61 & 0 & 0 \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
48 & -38 & 12 & 54 & 4 & 28 & 0 & 82 & 48 & -9 & 72 & 33 & 0 & 1
\end{array}$$

A.2 Revised Simplex Method

$$\vec{x}_B^* = \begin{pmatrix} 1 \\ 1 \\ 95 \end{pmatrix}, \quad \boldsymbol{C_{B_0}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 5 & 1 \end{pmatrix}$$

<u>Iteration 1:</u>

$$\begin{array}{lrcll}
1. & \vec{y}^T \boldsymbol{C_{B_0}} & = & \vec{b}_{B_0}^T & \\
 & \vec{y}^T & = & \begin{pmatrix} 133 & 125 & 0 \end{pmatrix} & \\
 & b_2 - \vec{y}^T \vec{c}_2 & = & 2, & \Rightarrow x_2 \text{ enters the basis} \\
2. & \boldsymbol{C_{B_0}} \vec{d} & = & \vec{c}_2 & \\
 & \vec{d} & = & \begin{pmatrix} 1 & 0 & 50 \end{pmatrix}^T & \\
 & \vec{x}_B^* - x_e \vec{d} & \geq & \vec{0} & \\
 & x_e & = & 1, & \Rightarrow x_1 \text{ leaves the basis} \\
3. & \vec{x}_B^* & = & \begin{pmatrix} 1 \\ 1 \\ 45 \end{pmatrix}, & \boldsymbol{C_{B_1}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 50 & 5 & 1 \end{pmatrix}
\end{array}$$

<u>Iteration 2:</u>

$$\begin{array}{lrcll}
1. & \vec{y}^T \boldsymbol{C_{B_1}} & = & \vec{b}_{B_1}^T & \\
 & \vec{y}^T & = & \begin{pmatrix} 135 & 125 & 0 \end{pmatrix} & \\
 & b_1 - \vec{y}^T \vec{c}_1 & = & -2 & b_3 - \vec{y}^T \vec{c}_3 = -2 \\
 & b_4 - \vec{y}^T \vec{c}_4 & = & 7, & \Rightarrow x_4 \text{ enters the basis} \\
2. & \boldsymbol{C_{B_1}} \vec{d} & = & \vec{c}_4 & \\
 & \vec{d} & = & \begin{pmatrix} 1 & 0 & -8 \end{pmatrix}^T & \\
 & \vec{x}_B^* - x_e \vec{d} & \geq & \vec{0} & \\
 & x_e & = & 1, & \Rightarrow x_2 \text{ leaves the basis} \\
3. & \vec{x}_B^* & = & \begin{pmatrix} 1 \\ 1 \\ 53 \end{pmatrix}, & \boldsymbol{C_{B_2}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 42 & 5 & 1 \end{pmatrix}
\end{array}$$

Iteration 3:

$$\begin{array}{lrcl}
1. & \vec{y}^T \boldsymbol{C_{B_2}} & = & \vec{b}^T_{B_2} \\
 & \vec{y}^T & = & \begin{pmatrix} 142 & 125 & 0 \end{pmatrix} \\
 & b_1 - \vec{y}^T\vec{c}_1 & = & -9 \qquad b_2 - \vec{y}^T\vec{c}_2 = -7 \\
 & b_3 - \vec{y}^T\vec{c}_3 & = & -9 \qquad b_5 - \vec{y}^T\vec{c}_5 = 0 \\
 & b_6 - \vec{y}^T\vec{c}_6 & = & 19, \qquad \Rightarrow x_6 \text{ enters the basis} \\
2. & \boldsymbol{C_{B_2}}\vec{d} & = & \vec{c}_6 \\
 & \vec{d} & = & \begin{pmatrix} 1 & 0 & -4 \end{pmatrix}^T \\
 & \vec{x}^*_B - x_e\vec{d} & \geq & \vec{0} \\
 & x_e & = & 1, \qquad \Rightarrow x_4 \text{ leaves the basis} \\
3. & \vec{x}^*_B & = & \begin{pmatrix} 1 \\ 1 \\ 57 \end{pmatrix}, \qquad \boldsymbol{C_{B_3}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 38 & 5 & 1 \end{pmatrix}
\end{array}$$

Iteration 4:

$$\begin{array}{lrcl}
1. & \vec{y}^T \boldsymbol{C_{B_3}} & = & \vec{b}^T_{B_3} \\
 & \vec{y}^T & = & \begin{pmatrix} 161 & 125 & 0 \end{pmatrix} \\
 & b_1 - \vec{y}^T\vec{c}_1 & = & -28 \qquad b_2 - \vec{y}^T\vec{c}_2 = -26 \\
 & b_3 - \vec{y}^T\vec{c}_3 & = & -28 \qquad b_4 - \vec{y}^T\vec{c}_4 = -19 \\
 & b_5 - \vec{y}^T\vec{c}_5 & = & -19 \\
 & b_7 - \vec{y}^T\vec{c}_7 & = & 8, \qquad \Rightarrow x_7 \text{ enters the basis} \\
2. & \boldsymbol{C_{B_3}}\vec{d} & = & \vec{c}_7 \\
 & \vec{d} & = & \begin{pmatrix} 0 & 1 & 91 \end{pmatrix}^T \\
 & \vec{x}^*_B - x_e\vec{d} & \geq & \vec{0} \\
 & x_e & = & \frac{57}{91}, \qquad \Rightarrow x_{13} \text{ leaves the basis} \\
3. & \vec{x}^*_B & = & \begin{pmatrix} 1 \\ \frac{34}{91} \\ \frac{57}{91} \end{pmatrix}, \qquad \boldsymbol{C_{B_4}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 38 & 5 & 96 \end{pmatrix}
\end{array}$$

<u>Iteration 5:</u>

1.
$$\begin{aligned}
\vec{y}^T \boldsymbol{C_{B_4}} &= \vec{b}_{B_4}^T \\
\vec{y}^T &= \begin{pmatrix} 157\frac{60}{91} & 124\frac{51}{91} & \frac{8}{91} \end{pmatrix} \\
b_1 - \vec{y}^T\vec{c}_1 &= -24\tfrac{60}{91} & b_2 - \vec{y}^T\vec{c}_2 &= -27\tfrac{5}{91} \\
b_3 - \vec{y}^T\vec{c}_3 &= -32\tfrac{68}{91} & b_4 - \vec{y}^T\vec{c}_4 &= -19\tfrac{32}{91} \\
b_5 - \vec{y}^T\vec{c}_5 &= -21\tfrac{6}{13} \\
b_8 - \vec{y}^T\vec{c}_8 &= 15\tfrac{90}{91}, & \Rightarrow x_8 &\text{ enters the basis}
\end{aligned}$$

2.
$$\begin{aligned}
\boldsymbol{C_{B_4}}\vec{d} &= \vec{c}_8 \\
\vec{d} &= \begin{pmatrix} 0 & \frac{34}{91} & \frac{57}{91} \end{pmatrix}^T \\
\vec{x}_B^* - x_e\vec{d} &\geq \vec{0} \\
x_e &= 1, & \Rightarrow x_9 &\text{ leaves the basis}
\end{aligned}$$

3.
$$\vec{x}_B^* = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \qquad \boldsymbol{C_{B_5}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 38 & 62 & 96 \end{pmatrix}$$

<u>Iteration 6:</u>

1.
$$\begin{aligned}
\vec{y}^T \boldsymbol{C_{B_5}} &= \vec{b}_{B_5}^T \\
\vec{y}^T &= \begin{pmatrix} 175\frac{9}{17} & 169\frac{12}{17} & -\frac{13}{34} \end{pmatrix} \\
b_1 - \vec{y}^T\vec{c}_1 &= -42\tfrac{9}{17} & b_2 - \vec{y}^T\vec{c}_2 &= -21\tfrac{7}{17} \\
b_3 - \vec{y}^T\vec{c}_3 &= -7\tfrac{6}{17} & b_4 - \vec{y}^T\vec{c}_4 &= -17\tfrac{8}{17} \\
b_5 - \vec{y}^T\vec{c}_5 &= -8\tfrac{5}{17} & b_9 - \vec{y}^T\vec{c}_9 &= -42\tfrac{27}{34} \\
b_{10} - \vec{y}^T\vec{c}_{10} &= 7\tfrac{3}{17}, & \Rightarrow x_{10} &\text{ enters the basis}
\end{aligned}$$

2.
$$\begin{aligned}
\boldsymbol{C_{B_5}}\vec{d} &= \vec{c}_{10} \\
\vec{d} &= \begin{pmatrix} 0 & \frac{5}{17} & \frac{12}{17} \end{pmatrix}^T \\
\vec{x}_B^* - x_e\vec{d} &\geq \vec{0} \\
x_e &= 0, & \Rightarrow x_7 &\text{ leaves the basis}
\end{aligned}$$

3.
$$\vec{x}_B^* = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \qquad \boldsymbol{C_{B_6}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 38 & 62 & 86 \end{pmatrix}$$

Iteration 7:

1.
$$\begin{aligned}
\vec{y}^T \boldsymbol{C_{B_6}} &= \vec{b}_{B_6}^T \\
\vec{y}^T &= \begin{pmatrix} 164\tfrac{1}{6} & 151\tfrac{1}{6} & -\tfrac{1}{12} \end{pmatrix} \\
b_1 - \vec{y}^T\vec{c}_1 &= -31\tfrac{1}{6} & \quad b_2 - \vec{y}^T\vec{c}_2 &= -25 \\
b_3 - \vec{y}^T\vec{c}_3 &= -23\tfrac{1}{2} & \quad b_4 - \vec{y}^T\vec{c}_4 &= -18\tfrac{2}{3} \\
b_5 - \vec{y}^T\vec{c}_5 &= -16\tfrac{2}{3} & \quad b_7 - \vec{y}^T\vec{c}_7 &= -10\tfrac{1}{6} \\
b_9 - \vec{y}^T\vec{c}_9 &= -25\tfrac{3}{4} & \quad b_{11} - \vec{y}^T\vec{c}_{11} &= -11\tfrac{1}{4} \\
b_{12} - \vec{y}^T\vec{c}_{12} &= 47, & \quad \Rightarrow x_{12} & \text{ enters the basis}
\end{aligned}$$

2.
$$\begin{aligned}
\boldsymbol{C_{B_6}}\vec{d} &= \vec{c}_{12} \\
\vec{d} &= \begin{pmatrix} 0 & 3 & -2 \end{pmatrix}^T \\
\vec{x}_B^* - x_e\vec{d} &\geq \vec{0} \\
x_e &= \tfrac{1}{3}, & \quad \Rightarrow x_8 & \text{ leaves the basis}
\end{aligned}$$

3.
$$\vec{x}_B^* = \begin{pmatrix} 1 \\ \frac{1}{3} \\ \frac{2}{3} \end{pmatrix}, \qquad \boldsymbol{C_{B_7}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 38 & 14 & 86 \end{pmatrix}$$

Iteration 8:

1.
$$\begin{aligned}
\vec{y}^T \boldsymbol{C_{B_7}} &= \vec{b}_{B_7}^T \\
\vec{y}^T &= \begin{pmatrix} 188\tfrac{35}{36} & 207\tfrac{11}{36} & -\tfrac{53}{72} \end{pmatrix} \\
b_1 - \vec{y}^T\vec{c}_1 &= -55\tfrac{35}{36} & \quad b_2 - \vec{y}^T\vec{c}_2 &= -17\tfrac{1}{6} \\
b_3 - \vec{y}^T\vec{c}_3 &= 11\tfrac{3}{4}, & \quad \Rightarrow x_3 & \text{ enters the basis}
\end{aligned}$$

2.
$$\begin{aligned}
\boldsymbol{C_{B_7}}\vec{d} &= \vec{c}_3 \\
\vec{d} &= \begin{pmatrix} 1 & -\tfrac{3}{4} & \tfrac{3}{4} \end{pmatrix}^T \\
\vec{x}_B^* - x_e\vec{d} &\geq \vec{0} \\
x_e &= \tfrac{8}{9}, & \quad \Rightarrow x_{10} & \text{ leaves the basis}
\end{aligned}$$

3.
$$\vec{x}_B^* = \begin{pmatrix} \frac{1}{9} \\ 1 \\ \frac{8}{9} \end{pmatrix}, \qquad \boldsymbol{C_{B_8}} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 38 & 14 & 92 \end{pmatrix}$$

Iteration 9:

1.
$$\begin{aligned}
\vec{y}^T \boldsymbol{C_{B_8}} &= \vec{b}_{B_8}^T \\
\vec{y}^T &= \left(180\tfrac{19}{27} \;\; 204\tfrac{7}{27} \;\; -\tfrac{14}{27}\right) \\
b_1 - \vec{y}^T\vec{c}_1 &= -47\tfrac{19}{27} & b_2 - \vec{y}^T\vec{c}_2 &= -19\tfrac{7}{9} \\
b_4 - \vec{y}^T\vec{c}_4 &= -16\tfrac{25}{27} & b_5 - \vec{y}^T\vec{c}_5 &= -4\tfrac{13}{27} \\
b_7 - \vec{y}^T\vec{c}_7 &= -21\tfrac{13}{27} & b_8 - \vec{y}^T\vec{c}_8 &= -26\tfrac{1}{9} \\
b_9 - \vec{y}^T\vec{c}_9 &= -76\tfrac{2}{3} & b_{10} - \vec{y}^T\vec{c}_{10} &= -15\tfrac{2}{3} \\
b_{11} - \vec{y}^T\vec{c}_{11} &= -43\tfrac{8}{9} \\
b_{13} - \vec{y}^T\vec{c}_{13} &= \tfrac{14}{27}, & &\Rightarrow \; x_{13} \text{ enters the basis}
\end{aligned}$$

2.
$$\begin{aligned}
\boldsymbol{C_{B_8}}\vec{d} &= \vec{c}_{13} \\
\vec{d} &= \left(-\tfrac{1}{54} \;\; 0 \;\; \tfrac{1}{54}\right)^T \\
\vec{x}_B^* - x_e\vec{d} &\geq \vec{0} \\
x_e &= 48, & &\Rightarrow \; x_3 \text{ leaves the basis}
\end{aligned}$$

3.
$$\vec{x}_B^* = \begin{pmatrix} 1 \\ 1 \\ 48 \end{pmatrix}, \qquad \boldsymbol{C_{B_9}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 38 & 14 & 1 \end{pmatrix}$$

Iteration 10:

1.
$$\begin{aligned}
\vec{y}^T \boldsymbol{C_{B_9}} &= \vec{b}_{B_9}^T \\
\vec{y}^T &= \left(161 \;\; 197 \;\; 0\right) \\
b_1 - \vec{y}^T\vec{c}_1 &= -28 & b_2 - \vec{y}^T\vec{c}_2 &= -26 \\
b_3 - \vec{y}^T\vec{c}_3 &= -28 & b_4 - \vec{y}^T\vec{c}_4 &= -19 \\
b_5 - \vec{y}^T\vec{c}_5 &= -19 & b_7 - \vec{y}^T\vec{c}_7 &= -64 \\
b_8 - \vec{y}^T\vec{c}_8 &= -51 & b_9 - \vec{y}^T\vec{c}_9 &= -72 \\
b_{10} - \vec{y}^T\vec{c}_{10} &= -53 & b_{11} - \vec{y}^T\vec{c}_{11} &= -61
\end{aligned}$$

B Code

B.1 Library Excerpts

Listing 13: common.h

```c
/**
 * Common header file for use within this project.
 **/

#pragma once

#include <stdio.h>    /* input/output operations */
#include <stdlib.h>   /* memory allocation */
#include <string.h>   /* string operations */

/* formulas used within this project: a << b = a * 2^b, a >> b = a / 2^b */

#define DLLEXPORT __declspec(dllexport)
#define ACTIVITY_SIZE 12
#define POOL_SIZE 16
#define MAX_ACTIVITIES 10000000
#define MAX_POOLS 1000
#define MAX_CANDIDATES 100
#define NUM_DIGITS 10
#define LENGTH 20
#define EPS1 1.0e-5
#define EPS2 1.0e-8
#define REF_PERIOD 50
#define ERR_ACTIVITY "Activity index out of bounds"
#define ERR_CANDIDATE_OF "Number of candidates overflow"
#define ERR_CANDIDATE_NP "No pool defined for this candidate"
#define ERR_POOL "Number of pools overflow"
#define ERR_MEMORY "Requested memory could not be allocated"
#define ERR_FILE_OPEN "File could not be opened"
#define ERR_FILE_CLOSE "File could not be closed"
#define ERR_FILE_NAME "File name could not be constructed"
#define env(x, y) (x - y < EPS2 && y - x < EPS2) ? 1 : 0   /* determines if "x" is in EPS2
    environment of integer number "y" */


typedef struct activity activity;
typedef struct pool pool;

struct activity
{
    int id;
    int benefit;
    int cost;
};
```

```
struct pool
{
    int idx;    /* "aidx" of first candidate */
    int numc;   /* number of candidates in this pool */
    int bmax;   /* offset to candidate activity with maximal benefit value */
    int cmin;   /* offset to candidate activity with minimal cost value */
};

activity *activities;   /* array of activities on this route */
int *rids;              /* array of result ids (first element is size, second is sum of
    benefits) */
pool *pools;            /* array of pools on this route */
int **basis;            /* array of pool index / offset pairs */
double *xB;             /* array holding the basis variables' values */
double *y;              /* array representing vector "y" */
double *d;              /* array representing vector "d" */
int *sB;                /* array containing indices of "basis" to avoid searching */
int *best;              /* array of ids comprising the best solution */

int aidx, ridx, pidx;   /* current maximal array indices of "activities", "rids" and
    "pools" */

int abenefit, rbenefit;   /* sum of actual/result benefit values */
int acostl, rcostl;       /* actual/result cost level values (route cost level := route max
    cost - route cost) */
```

Listing 14: insertsa

```
/**
 * Inserts an activity of type "single".
 */
extern DLLEXPORT void insertsa(const int id, int benefit, int cost)
{
    if (cost < 0)
    {
        cost = 0;
    }
    if (benefit < min_benefit)
    {
        benefit = min_benefit;
    }
    else if (benefit > min_benefit << 1)
    {
        benefit = min_benefit << 1;
    }

    if (deltah != 0)
    {
        if (acostl >= 0)
        {
            trigger();
        }
        if (print)
        {
            printr();
        }

        reset();
    }

    aidx++;
    if (aidx >= num_activities)
    {
        leave(ERR_ACTIVITY);
    }

    (activities + aidx)->id = id;
    (activities + aidx)->benefit = benefit;
    (activities + aidx)->cost = cost;

    abenefit += benefit;
    acostl -= cost;

    cpool = NULL;
    nodes++;
}
```

Listing 15: insertpa

```
/**
 * Inserts an activity of type "pool".
 */
extern DLLEXPORT void insertpa()
{
    if (deltah != 0)
    {
        if (acostl >= 0)
        {
            trigger();
        }

        if (print)
        {
            printr();
        }

        reset();
    }

    pidx++;
    if (pidx >= num_pools)
    {
        leave(ERR_POOL);
    }
    cpool = pools + pidx;
    cpool->numc = 0;
    nodes++;
}
```

Listing 16: insertca

```
/**
 * Inserts an activity of type "candidate".
 */
extern DLLEXPORT void insertca(const int id, int benefit, int cost)
{
    if (cost < 0)
    {
        cost = 0;
    }
    if (benefit < min_benefit)
    {
        benefit = min_benefit;
    }
    else if (benefit > min_benefit << 1)
    {
        benefit = min_benefit << 1;
    }

    aidx++;
    if (aidx >= num_activities)
    {
        leave(ERR_ACTIVITY);
    }
    if (cpool == NULL)
    {
        leave(ERR_CANDIDATE_NP);
    }

    (activities + aidx)->id = id;
    (activities + aidx)->benefit = benefit;
    (activities + aidx)->cost = cost;

    cpool->numc++;
    if (cpool->numc == 1)
    {
        cpool->idx = aidx;
        cpool->bmax = 0;
        cpool->cmin = 0;
        maxb = activities + aidx;
        minc = activities + aidx;
    }
    else if (cpool->numc > 1 && cpool->numc <= num_candidates)
    {
        if (benefit > maxb->benefit || (benefit == maxb->benefit && cost < maxb->cost))
        {
            cpool->bmax = aidx - cpool->idx;
            maxb = activities + aidx;
        }
        if (cost < minc->cost || (cost == minc->cost && benefit > minc->benefit))
        {
            cpool->cmin = aidx - cpool->idx;
            minc = activities + aidx;
        }
    }
    else
    {
        leave(ERR_CANDIDATE_OF);
    }
}
```

Listing 17: trigger

```
/**
 * Triggers the processing of the actual route.
 */
void trigger()
{
    int i;

    lbenefit = rbenefit - abenefit;
    lcost = acostl - rcostl;

    /* is a shortcut possible? */
    i = shortcut();
    if (i == -2 || i == 1)
    {
        return;
    }
    else if (i == 0)
    {
        if (simplex() == 0)
        {
            (void)eval();    /* return type not needed here */
        }
    }

    if (abenefit + lbenefit < rbenefit || abenefit + lbenefit == rbenefit && acostl - lcost
        <= rcostl)
    {
        /* solution not a global optimum */
        return;
    }
    rbenefit = abenefit + lbenefit;
    rcostl = acostl - lcost;
    fillr();
}
```

Listing 18: step0

```
/**
 * Fills "basis", "sB" and "xB".
 */
void step0()
{
    pool *p;
    int i;

    bidx = 0;
    for (i = 0; i <= pidx; i++)
    {
        if ((p = pools + i)->numc == 0)
        {
            sB[i] = -1;
            continue;
        }

        basis[0][bidx] = i;
        basis[1][bidx] = p->cmin;
        sB[i] = bidx;
        xB[bidx] = 1;
        bidx++;
    }

    /* artificial slack variable */
    basis[0][bidx] = bidx;
    basis[1][bidx] = bidx;
    sB[i] = bidx;
    xB[bidx] = acostl - minc;

    /* init indices of basis variables from the same pool with the index position of the
        artificial slack variable */
    idx1 = bidx;
    idx2 = bidx;
}
```

B.2 Wrapper Excerpts

Listing 19: Operator.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.InteropServices;   // DLL support

namespace WorkflowCompositionWrapper
{
    public class Operator
    {
        private static Operator op;
        private delegate void Exception(string message);

        private Exception exception;

        private byte exceptionType;

        [DllImport("WorkflowCompositionLibrary.dll")]
        private static extern void init(int num_activities, int num_pools,
            int num_candidates, int max_cost, int min_benefit, Exception exception,
            byte print);

        [DllImport("WorkflowCompositionLibrary.dll")]
        private static extern void refresh(int max_cost, int min_benefit, byte print);

        [DllImport("WorkflowCompositionLibrary.dll")]
        private static extern void insertsa(int id, int benefit, int cost);

        [DllImport("WorkflowCompositionLibrary.dll")]
        private static extern void insertpa();

        [DllImport("WorkflowCompositionLibrary.dll")]
        private static extern void insertca(int id, int benefit, int cost);

        [DllImport("WorkflowCompositionLibrary.dll")]
        private static extern void insertbs(int steps);

        [DllImport("WorkflowCompositionLibrary.dll")]
        private static extern IntPtr result();

        [DllImport("WorkflowCompositionLibrary.dll")]
        private static extern void clean();


        private Operator(int num_activities, int num_pools, int num_candidates, int max_cost,
             int min_benefit, bool print)
        {
            exceptionType = 2;

            exception += new Exception(ThrowException);

            if (print)
            {
                init(num_activities, num_pools, num_candidates, max_cost, min_benefit,
                    exception, 1);
            }
            else
            {
                init(num_activities, num_pools, num_candidates, max_cost, min_benefit,
                    exception, 0);
            }
            exceptionType = 0;
        }
```

```
private void ThrowException(string message)
{
    exception += new Exception(ThrowException);

    switch (exceptionType)
    {
        case 0:
            throw new InsertActivityException(message);
        case 1:
            exceptionType = 0;
            throw new ReturnResultException(message);
        default:
            exceptionType = 0;
            throw new System.Exception(message);
    }
}

public static Operator CreateInstance(int num_activities, int num_pools,
    int num_candidates, int max_cost, int min_benefit, bool print)
{
    return (op == null) ? (op = new Operator(num_activities, num_pools,
        num_candidates, max_cost, min_benefit, print)) : null;
}

public void RefreshOperator(int max_cost, int min_benefit, bool print)
{
    exceptionType = 2;

    if (print)
    {
        refresh(max_cost, min_benefit, 1);
    }
    else
    {
        refresh(max_cost, min_benefit, 0);
    }

    exceptionType = 0;
}

public void InsertSingleActivity(int id, int benefit, int cost)
{
    insertsa(id, benefit, cost);
}

public void InsertPoolActivity()
{
    insertpa();
}
```

```
        public void InsertCandidateActivity(int id, int benefit, int cost)
        {
            insertca(id, benefit, cost);
        }

        public void InsertBackStatement(int steps)
        {
            insertbs(steps);
        }

        public int[] ReturnResult()
        {
            exceptionType = 1;

            int[] intArr = null;
            IntPtr intPtr = result();
            if (intPtr != null)
            {
                int length = Marshal.ReadInt32(intPtr);
                if (length > 0)
                {
                    intArr = new int[length];
                    for (int i = 0; i < length; i++)
                    {
                        intArr[i] = Marshal.ReadInt32(intPtr, (i + 1) * IntPtr.Size);
                    }
                }
            }

            exceptionType = 0;

            return intArr;
        }

        public void MakeClean()
        {
            exceptionType = 2;

            clean();

            exceptionType = 0;
            exception -= new Exception(ThrowException);
            op = null;
        }
    }
}
```

B.3 Test Excerpts

Listing 20: InsertActivitySITest.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using WorkflowCompositionWrapper;
using NUnit.Framework;

namespace WorkflowCompositionTest
{
    /// <summary>
    /// Class for testing "InsertActivity" methods with small instances.
    /// </summary>
    [TestFixture]
    public class InsertActivitySITest
    {
        private Operator op;

        [TestFixtureSetUp]
        public void Init()
        {
            op = Operator.CreateInstance(35, 5, 5, 0, 0, true);
        }

        /// <summary>
        /// No activity.
        /// </summary>
        [Test]
        public void Test00()
        {
            try
            {
                op.RefreshOperator(50, 100, true);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Empty Pool activity.
        /// </summary>
        [Test]
        public void Test01()
        {
            try
            {
                op.RefreshOperator(0, 0, true);
                op.InsertPoolActivity();
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Single activity.
        /// </summary>
        [Test]
        public void Test02()
        {
            try
            {
                op.RefreshOperator(0, 1, true);


                op.InsertSingleActivity(0, 1, 0);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Pool activity with one candidate activity.
        /// </summary>
        [Test]
        public void Test03()
        {
            try
            {
                op.RefreshOperator(0, 1, true);


                op.InsertPoolActivity();
                op.InsertCandidateActivity(0, 1, 0);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Pool activity with five candidate activities.
        /// </summary>
        [Test]
        public void Test04()
        {
            try
            {
                op.RefreshOperator(0, 9, true);


                op.InsertPoolActivity();
                op.InsertCandidateActivity(int.MinValue, 0, 0);
                op.InsertCandidateActivity(-1, 9, 4);
                op.InsertCandidateActivity(0, 18, 56);
                op.InsertCandidateActivity(1, 5000, 1234);
                op.InsertCandidateActivity(int.MaxValue, int.MaxValue, int.MaxValue);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Empty pool activity and one single activity.
        /// </summary>
        [Test]
        public void Test05()
        {
            try
            {
                op.RefreshOperator(0, 1, true);


                op.InsertPoolActivity();

                op.InsertSingleActivity(1, 1, 0);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Single activity and one empty pool activity.
        /// </summary>
        [Test]
        public void Test06()
        {
            try
            {
                op.RefreshOperator(0, 1, true);


                op.InsertSingleActivity(0, 1, 0);

                op.InsertPoolActivity();
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Single route (1).
        /// </summary>
        [Test]
        public void Test07()
        {
            try
            {
                op.RefreshOperator(0, 7, true);


                op.InsertPoolActivity();

                op.InsertSingleActivity(0, 1, 0);

                op.InsertPoolActivity();
                op.InsertCandidateActivity(1, 7, 5);
                op.InsertCandidateActivity(2, 14, 23);

                op.InsertSingleActivity(0, 3, 1);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Single route (2).
        /// </summary>
        [Test]
        public void Test08()
        {
            try
            {
                op.RefreshOperator(0, 84, true);


                op.InsertPoolActivity();
                op.InsertCandidateActivity(0, 0, 0);
                op.InsertCandidateActivity(1, 123, 78);

                op.InsertSingleActivity(2, 84, 44);

                op.InsertPoolActivity();
                op.InsertPoolActivity();
                op.InsertCandidateActivity(3, 3456, 567);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Pool activity with two children.
        /// </summary>
        [Test]
        public void Test09()
        {
            try
            {
                op.RefreshOperator(0, 10, true);


                op.InsertPoolActivity();
                op.InsertCandidateActivity(0, 1, 0);

                op.InsertPoolActivity();
                op.InsertCandidateActivity(1, 15, 13);

                op.InsertBackStatement(1);

                op.InsertSingleActivity(2, 0, 4);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Single activity with two children.
        /// </summary>
        [Test]
        public void Test10()
        {
            try
            {
                op.RefreshOperator(0, 350, true);


                op.InsertSingleActivity(0, 1, 0);
                op.InsertSingleActivity(1, 6, 2);

                op.InsertBackStatement(1);

                op.InsertPoolActivity();
                op.InsertCandidateActivity(2, 623, 391);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Branching on pool activity.
        /// </summary>
        [Test]
        public void Test11()
        {
            try
            {
                op.RefreshOperator(200, 54, true);


                op.InsertSingleActivity(0, 77, 49);

                op.InsertPoolActivity();
                op.InsertCandidateActivity(1, 54, 50);

                op.InsertSingleActivity(2, 112, 89);
                op.InsertSingleActivity(3, 85, 60);

                op.InsertBackStatement(2);

                op.InsertPoolActivity();
                op.InsertCandidateActivity(4, 99, 91);

                op.InsertSingleActivity(5, 1, 0);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Branching on single activity.
        /// </summary>
        [Test]
        public void Test12()
        {
            try
            {
                op.RefreshOperator(1675, 204, true);


                op.InsertPoolActivity();
                op.InsertCandidateActivity(0, 403, 299);
```

```
                op.InsertCandidateActivity(1, 239, 187);

                op.InsertSingleActivity(2, 367, 350);

                op.InsertPoolActivity();
                op.InsertCandidateActivity(3, 309, 268);

                op.InsertSingleActivity(4, 356, 323);
                op.InsertSingleActivity(5, 375, 341);
                op.InsertSingleActivity(6, 234, 206);

                op.InsertBackStatement(4);

                op.InsertSingleActivity(7, 299, 272);
                op.InsertSingleActivity(8, 263, 250);
                op.InsertSingleActivity(9, 204, 234);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Branching on two nodespr.
        /// </summary>
        [Test]
        public void Test13()
        {
            try
            {
                op.RefreshOperator(35, 6, true);


                op.InsertSingleActivity(0, 1, 0);
                op.InsertSingleActivity(1, 2, 1);
                op.InsertSingleActivity(2, 3, 2);
                op.InsertSingleActivity(3, 2, 3);

                op.InsertPoolActivity();
                op.InsertCandidateActivity(4, 11, 10);
                op.InsertCandidateActivity(5, 12, 11);

                op.InsertSingleActivity(6, 4, 5);
                op.InsertSingleActivity(7, 5, 6);
                op.InsertSingleActivity(8, 6, 7);

                op.InsertBackStatement(3);

                op.InsertPoolActivity();
                op.InsertCandidateActivity(9, 111, 110);
                op.InsertCandidateActivity(10, 112, 111);
                op.InsertSingleActivity(11, 8, 9);

                op.InsertBackStatement(5);

                op.InsertSingleActivity(12, 9, 10);
                op.InsertSingleActivity(13, 10, 11);
                op.InsertSingleActivity(14, 11, 12);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Equal activities.
        /// </summary>
        [Test]
        public void Test14()
        {
            try
            {
                op.RefreshOperator(200, 100, true);


                op.InsertSingleActivity(0, 100, 50);

                op.InsertPoolActivity();
                op.InsertCandidateActivity(1, 100, 50);
                op.InsertCandidateActivity(2, 100, 50);

                op.InsertSingleActivity(3, 100, 50);

                op.InsertPoolActivity();
                op.InsertCandidateActivity(4, 100, 50);
                op.InsertCandidateActivity(5, 100, 50);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// min_benefit = 0.
        /// </summary>
        [Test]
        public void Test15()
        {
            try
            {
                op.RefreshOperator(200, 0, true);
```

```
                op.InsertPoolActivity();
                op.InsertCandidateActivity(0, 13, 25);
                op.InsertCandidateActivity(1, -3, 75);

                op.InsertSingleActivity(2, 18, 50);

                op.InsertPoolActivity();
                op.InsertCandidateActivity(3, 56, 75);
                op.InsertCandidateActivity(4, 21, 25);

                op.InsertSingleActivity(5, 88, 50);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Debugged test instance (1).
        /// </summary>
        [Test]
        public void TestD1()
        {
            try
            {
                op.RefreshOperator(35, 13, false);

                op.InsertPoolActivity();
                op.InsertCandidateActivity(0, 2, 0);
                op.InsertCandidateActivity(1, 13, 12);
                op.InsertCandidateActivity(2, 20, 17);

                op.InsertPoolActivity();
                op.InsertCandidateActivity(3, 19, 16);
                op.InsertCandidateActivity(4, 0, 0);
                op.InsertCandidateActivity(5, 25, 20);

                op.InsertPoolActivity();
                op.InsertCandidateActivity(6, 0, 1);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Worst case of accuracy for max_cost and min_benefit.
        /// </summary>
        [Test]
        public void TestA1()
        {
            try
            {
                op.RefreshOperator(100, 100, false);

                op.InsertPoolActivity();
                op.InsertCandidateActivity(0, 100, 0);
                op.InsertCandidateActivity(1, 101, 1);

                op.InsertPoolActivity();
                op.InsertCandidateActivity(2, 100, 0);
                op.InsertCandidateActivity(3, 200, 100);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        [TestFixtureTearDown]
        public void Clean()
        {
            try
            {
                // wrapper instance is null now
                op.MakeClean();
                // this instance is null now
                op = null;
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        /// <summary>
        /// Gets the operator for use within other test files.
        /// </summary>
        public Operator Op
        {
            get { return op; }
        }
    }
}
```

Listing 21: ReturnResultSITest.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using WorkflowCompositionWrapper;
using NUnit.Framework;
```

```
namespace WorkflowCompositionTest
{
    /// <summary>
    /// Class for testing "ReturnResult()" with small instances.
    /// </summary>
    [TestFixture]
    public class ReturnResultSITest
    {
        private InsertActivitySITest iasit;

        public ReturnResultSITest()
        {
            iasit = new InsertActivitySITest();
        }

        [TestFixtureSetUp]
        public void Init()
        {
            iasit.Init();
        }

        [Test]
        public void Test00()
        {
            try
            {
                iasit.Test00();
                int[] rids = iasit.Op.ReturnResult();

                Program.CopyFiles("si", "00");

                Assert.IsNull(rids);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        [Test]
        public void Test01()
        {
            try
            {
                iasit.Test01();
                int[] rids = iasit.Op.ReturnResult();

                Program.CopyFiles("si", "01");

                Assert.IsNull(rids);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        [Test]
        public void Test02()
        {
            try
            {
                iasit.Test02();
                int[] rids = iasit.Op.ReturnResult();
                Program.CopyFiles("si", "02");

                Assert.AreEqual(2, rids.Length);
                Assert.AreEqual(1, rids[0]);
                Assert.AreEqual(0, rids[1]);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        [Test]
        public void Test03()
        {
            try
            {
                iasit.Test03();
                int[] rids = iasit.Op.ReturnResult();
                Program.CopyFiles("si", "03");

                Assert.AreEqual(2, rids.Length);
                Assert.AreEqual(1, rids[0]);
                Assert.AreEqual(0, rids[1]);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        [Test]
        public void Test04()
        {
            try
            {
                iasit.Test04();
                int[] rids = iasit.Op.ReturnResult();
                Program.CopyFiles("si", "04");

                Assert.AreEqual(2, rids.Length);
                Assert.AreEqual(9, rids[0]);
                Assert.AreEqual(int.MinValue, rids[1]);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        [Test]
        public void Test05()
        {
            try
            {
                iasit.Test05();
                int[] rids = iasit.Op.ReturnResult();
                Program.CopyFiles("si", "05");

                Assert.AreEqual(2, rids.Length);
                Assert.AreEqual(1, rids[0]);
                Assert.AreEqual(1, rids[1]);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        [Test]
        public void Test06()
        {
            try
            {
                iasit.Test06();
                int[] rids = iasit.Op.ReturnResult();
                Program.CopyFiles("si", "06");

                Assert.AreEqual(2, rids.Length);
                Assert.AreEqual(1, rids[0]);
                Assert.AreEqual(0, rids[1]);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        [Test]
        public void Test07()
        {
            try
            {
                iasit.Test07();
                int[] rids = iasit.Op.ReturnResult();
                Program.CopyFiles("si", "07");

                Assert.IsNull(rids);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        [Test]
        public void Test08()
        {
            try
            {
                iasit.Test08();
                int[] rids = iasit.Op.ReturnResult();
                Program.CopyFiles("si", "08");

                Assert.IsNull(rids);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        [Test]
        public void Test09()
        {
            try
            {
                iasit.Test09();
                int[] rids = iasit.Op.ReturnResult();
                Program.CopyFiles("si", "09");

                Assert.IsNull(rids);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        [Test]
        public void Test10()
        {
            try
            {
                iasit.Test10();
                int[] rids = iasit.Op.ReturnResult();
                Program.CopyFiles("si", "10");
                Assert.IsNull(rids);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        [Test]
        public void Test11()
        {
            try
            {
                iasit.Test11();
                int[] rids = iasit.Op.ReturnResult();
                Program.CopyFiles("si", "11");

                Assert.AreEqual(5, rids.Length);
                Assert.AreEqual(284, rids[0]);
                Assert.AreEqual(0, rids[1]);
                Assert.AreEqual(1, rids[2]);
                Assert.AreEqual(4, rids[3]);
                Assert.AreEqual(5, rids[4]);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }

        [Test]
        public void Test12()
        {
            try
            {
                iasit.Test12();
                int[] rids = iasit.Op.ReturnResult();
                Program.CopyFiles("si", "12");

                Assert.AreEqual(7, rids.Length);
                Assert.AreEqual(1880, rids[0]);
                Assert.AreEqual(1, rids[1]);
                Assert.AreEqual(2, rids[2]);
                Assert.AreEqual(3, rids[3]);
                Assert.AreEqual(4, rids[4]);
                Assert.AreEqual(5, rids[5]);
                Assert.AreEqual(6, rids[6]);
            }
            catch (Exception e)
            {
                Assert.Fail(e.Message);
            }
        }
```

```
[Test]
public void Test13()
{
    try
    {
        iasit.Test13();
        int[] rids = iasit.Op.ReturnResult();
        Program.CopyFiles("si", "13");

        Assert.AreEqual(9, rids.Length);
        Assert.AreEqual(54, rids[0]);
        Assert.AreEqual(0, rids[1]);
        Assert.AreEqual(1, rids[2]);
        Assert.AreEqual(2, rids[3]);
        Assert.AreEqual(3, rids[4]);
        Assert.AreEqual(5, rids[5]);
        Assert.AreEqual(6, rids[6]);
        Assert.AreEqual(7, rids[7]);
        Assert.AreEqual(8, rids[8]);
    }
    catch (Exception e)
    {
        Assert.Fail(e.Message);
    }
}

[Test]
public void Test14()
{
    try
    {
        iasit.Test14();
        int[] rids = iasit.Op.ReturnResult();
        Program.CopyFiles("si", "14");

        Assert.AreEqual(5, rids.Length);
        Assert.AreEqual(400, rids[0]);
        Assert.AreEqual(0, rids[1]);
        Assert.AreEqual(1, rids[2]);
        Assert.AreEqual(3, rids[3]);
        Assert.AreEqual(4, rids[4]);
    }
    catch (Exception e)
    {
        Assert.Fail(e.Message);
    }
}

[Test]
public void Test15()
{
    try
```

```
        {
            iasit.Test15();
            int[] rids = iasit.Op.ReturnResult();
            Program.CopyFiles("si", "15");

            Assert.AreEqual(5, rids.Length);
            Assert.AreEqual(0, rids[0]);
            Assert.AreEqual(0, rids[1]);
            Assert.AreEqual(2, rids[2]);
            Assert.AreEqual(4, rids[3]);
            Assert.AreEqual(5, rids[4]);
        }
        catch (Exception e)
        {
            Assert.Fail(e.Message);
        }
    }

    [Test]
    public void TestD1()
    {
        try
        {
            iasit.TestD1();
            int[] rids = iasit.Op.ReturnResult();
            Program.CopyFiles("si", "d1");

            Assert.AreEqual(4, rids.Length);
            Assert.IsTrue((52.0 - rids[0]) / rids[0] < 1.0 / (rids.Length - 1));
        }
        catch (Exception e)
        {
            Assert.Fail(e.Message);
        }
    }

    [Test]
    public void TestA1()
    {
        try
        {
            iasit.TestA1();
            int[] rids = iasit.Op.ReturnResult();
            Program.CopyFiles("si", "a1");

            Assert.AreEqual(3, rids.Length);
            Assert.AreEqual(201, rids[0]);
            Assert.AreEqual(1, rids[1]);
            Assert.AreEqual(2, rids[2]);
        }
        catch (Exception e)
        {
            Assert.Fail(e.Message);
            }
        }

        [TestFixtureTearDown]
        public void Clean()
        {
            iasit.Clean();
        }
    }
}
```

C Measurements

	α[E-7s]	β[E-6s]	γ[E-7s]	δ[E-9s]	ε[E-9s]	ζ[E-2s]
Test01	2.422	1.359	8.559	1.908	1.007	1.173
Test02	2.426	1.878	7.741	0.903	1.030	0.867
Test03	2.422	-0.476	8.226	1.870	1.013	1.218
Test04	2.425	3.277	9.704	2.282	0.984	0.717
Test05	2.431	1.734	9.963	3.126	0.971	0.569
Test06	2.419	2.230	9.298	1.003	1.013	0.985
Test07	2.417	-0.956	8.532	2.547	1.000	1.152
Test08	2.419	4.522	8.240	0.773	1.026	0.924
Test09	2.420	1.517	8.440	2.346	1.005	0.978
Test10	2.420	3.107	8.917	2.281	0.994	1.016
Average	2.422	1.819	8.762	1.904	1.004	0.960

Table 4: Coefficients of the runtime equation for s : 0[1.1E+6]9.9E+6, p : 0[100]1000 and c : 0[50]100

	β[E-6s]	γ[E-6s]	δ[E-9s]	ε[E-9s]
Test01	3.537	0.942	2.563	0.993
Test02	3.391	0.852	2.452	1.003
Test03	3.846	0.878	2.697	0.997
Test04	3.611	0.960	2.865	0.976
Test05	5.091	1.054	3.219	0.966
Test06	4.084	0.972	3.035	0.979
Test07	4.833	1.046	2.235	0.981
Test08	4.708	0.929	2.250	0.994
Test09	4.270	1.054	2.534	0.978
Test10	3.789	0.838	2.329	1.007
Average	4.116	0.952	2.618	0.987

Table 5: Improved values for β, γ, δ and ε with s : 0[1]0, p : 0[10]1000 and c : 0[10]100

	$\boldsymbol{\beta}$[E-6s]	$\boldsymbol{\gamma}$[E-7s]	$\boldsymbol{\delta}$[E-9s]	$\boldsymbol{\varepsilon}$[E-9s]
Test01	4.411	9.630	2.761	0.984
Test02	0.376	9.680	2.741	0.983
Test03	2.629	8.945	2.658	0.993
Test04	3.110	9.303	2.785	0.990
Test05	5.236	7.481	1.831	1.025
Test06	4.374	8.954	2.497	1.000
Test07	5.701	9.935	2.684	0.980
Test08	1.798	9.206	2.802	0.995
Test09	4.715	9.147	2.270	0.995
Test10	4.534	9.017	2.399	1.000
Average	3.688	9.130	2.543	0.994

Table 6: Cuts of the runtime function for $s = 495\,000$ * Test Number

	s	**p**	**c**	$\boldsymbol{\zeta}$[E-1s]	$\boldsymbol{\zeta/\langle r\rangle}$[E-2]
Test01	7671469	172	6	0.739	3.823
Test02	4186046	972	91	0.241	1.137
Test03	9730246	473	98	1.404	5.052
Test04	5710944	397	6	0.404	2.814
Test05	7373143	313	86	0.563	2.882
Test06	8972702	176	11	0.643	2.867
Test07	3884756	770	51	0.317	2.355
Test08	5290681	329	42	0.313	2.280
Test09	8496313	288	9	0.669	3.133
Test10	370513	317	13	0.060	5.303
Average				0.535	3.164

Table 7: Estimating ζ and $\zeta/\langle r\rangle$ for $\alpha = 2.42\text{E-}7s$, $\beta = 3.7\text{E-}6s$, $\gamma = 9.2\text{E-}7s$, $\delta = 2.55\text{E-}9s$ and $\varepsilon = 9.94\text{E-}10s$

Bibliography

[AAC07] J. Anselmi, D. Ardagna, and P. Cremonesi. A QoS-based Selection Approach of Autonomic Grid Services. In *SOCP '07: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*, pages 1–8, New York, NY, USA, 2007. ACM.

[ABCC98] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. On the Solution of Traveling Salesman Problems. In *Doc. Math. J. DMV III Extra Volume*, pages 645–656. ICM, 1998.

[BS85] I.N. Bronstein and K.A. Semendjajew. *Taschenbuch der Mathematik*. Gemeinschaftsausgabe Verlag Nauka, Moscow and BSB B.G.Teubner Verlagsgesellschaft, Leipzig, Moscow and Leipzig, 1985.

[CDPEV05] G. Canfora, M. Di Penta, R. Esposito, and M.L. Villani. An Approach for QoS-aware Service Composition based on Genetic Algorithms. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1069–1075, New York, NY, USA, 2005. ACM.

[Chv83] V. Chvátal. *Linear Programming*. W.H. Freeman and Company, New York, 1983.

[Dem03] W. Demtröder. *Experimentalphysik 1*. Springer-Verlag, Berlin, 2003.

[Gol91] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[JRGM04] M.C. Jaeger, G. Rojec-Goldmann, and G. Muhl. QoS Aggregation for Web Service Composition using Workflow Patterns. *Enterprise Distributed Object Computing Conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International*, pages 149–159, Sept. 2004.

[KM72] V. Klee and G. Minty. How good is the simplex algorithm? In *Inequalities*, volume III, pages 159–175. O. Shisha, Ed. Academic Press, 1972.

[KMN03] S. Keserovic, D. Mortenson, and A. Nathan. An Overview of Managed/Unmanaged Code Interoperability, October 2003. http://msdn.microsoft.com/en-us/library/ms973872.aspx.

[MCD08] D.A. Menascé, E. Casalicchio, and V. Dubey. A Heuristic Approach to Optimal Service Selection in Service Oriented Architectures. In *WOSP '08: Proceedings of the 7th international workshop on Software and performance*, pages 13–24, New York, NY, USA, 2008. ACM.

[Mur81] B.A. Murtagh. *Advanced Linear Programming: Computation and Practice*. McGraw-Hill, New York, 1981.

[NW88] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, New York, 1988.

[Ove97] M.L. Overton. Linear Programming. Draft for Encyclopedia Americana, December 1997. http://www.cs.nyu.edu/overton/g22_lp/encyc/article_web.html.

[Sed83] R. Sedgewick. *Algorithms*. Addison-Wesley, Reading MA, 1983.

[Van01] R.J. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer, New York, 2001.

[Vie03] R. Viertl. *Einführung in die Stochastik*. Springer-Verlag, Wien, 2003.

[YTX$^+$07] Y. Yang, S. Tang, Y. Xu, W. Zhang, and L. Fang. An Approach to QoS-aware Service Selection in Dynamic Web Service Composition. In *ICNS '07: Proceedings of the Third International Conference on Networking and Services*, pages 18–36, Washington, DC, USA, June 2007. IEEE Computer Society.

[YZL07] T. Yu, Y. Zhang, and K.-J. Lin. Efficient Algorithms for Web Services Selection with End-to-End QoS Constraints. *ACM Trans. Web*, 1(1):6, 2007.